Программирование на Clojure
(для любопытных)

Содержание

…..Краткое введение

Clojure динамичный язык, реализованный на виртуальной машине Java (JVM). В своей основе язык имеет все необходимые средства для программирования в функциональном стиле: можно создавать чистые (без побочных эффектов) функции, имеет неизменяемые структурные типы и так далее. Но кроме того, Clojure обладает набором обычных свойств, характерных для императивных языков. Впрочем, совсем без использования изменяемых переменных и без побочных эффектов не обходятся даже и самые строго функциональные языки, вроде Haskell. Clojure – не объектно-ориентированный язык, хотя в нём имеются

некоторые, впрочем слабо выраженные, элементы, позволяющие реализовать такие свойства, как полиморфизм, наследование, создание пользовательских типов и тому подобное. Всё это базируется на Java, а потому очень громоздко и запутанно. Разбираться в нагромождении этих приёмов — большая скука.

Язык Clojure имеет довольно своеобразный синтаксис, который при первом знакомстве кажется необычным. Все выражения представлены в виде блоков, заключённых в круглые скобки. Для обозначения границ блоков не применяются ни фигурные скобки, ни ***begin-end*** или ***do-end***, как во всех прочих языках. По этой причине в программном коде всегда присутствует обилие круглых скобок и на первых порах программы кажутся плохо «читабельными». Многое, конечно, зависит от опыта программиста и при хорошей организации кода трудности восприятия преодолеваются.

Как и прочие языки, Clojure имеет интерактивный режим (Repl) для оперативного выполнения фрагментов кода в процессе разработки программы. Если дополнительно установить программу ***leiningen***, то для вызова режима Repl достаточно в командной строке выполнить команду:
lein repl
после чего появится приглашение в виде
user =>
после чего можно вводить программный код, который будет немедленно выполнен с выводом результата на экран. Слово ***user*** обозначает название области имён, активной на данный момент (***user*** по умолчанию). В процессе работы могут активироваться области с другими именами.

Файлы с программами на Clojure должны иметь расширение ***.clj***. Такие файлы тоже можно выполнять в Repl, для этого достаточно выполнить такую команду:
(load-file "c:/clo/temp.clj")
Здесь ***c:/clo/temp.clj*** – пример полного пути для выполняемого файла ***temp.clj.*** Можно указывать и относительный путь, если Repl открыт в соответствующей директории.

В языке присутствует огромное количество встроенных функций, в режиме Repl можно получить краткую справку о каждой из них с помощью команды:
(doc name) **-** здесь ***name*** — имя функции.

Можно даже получить программный код для заданной функции с помощью команды:

(source name)

При отсутствии функции с заданным именем возвращается ***nil***.

Clojure имеет свой IDE (на базе известного редактора Eclipse), называемый Counterclockwise, который позволяет работать как с исходными файлами, имеющими в названии расширение ***.clj***, так и в режиме Repl.

В заключение этой небольшой вводной части рассмотрим пример простейшей программы на Clojure (вместо надоевшей уже helloworld):

```
(defn f [x]
  (/(apply + x)(count x)))
```

Из текста нашей небольшой программы можно вывести достаточно много различной информации. Как нетрудно догадаться, ключевое слово ***defn*** объявляет новую функцию под названием ***f***.

Замечание: обычно рекомендуется для функций и переменных использовать идентификаторы, которые отражают смысл и содержание этих объектов, например, если функция вычисляет абсолютное значение числа, то её надо назвать ***absolute-value,*** или ***absoluteValue*** и тому подобное. Определённый смысл в этом есть только тогда, когда это «долго живущие» объекты, например названия библиотечных модулей. Если вводимый нами объект существует лишь на нескольких строках программы, названия должны быть как можно короче, поскольку длинные слова только ухудшают читабельность и ничего не дают. Следуя этому правилу, я всегда буду применять самые короткие идентификаторы, например, в наших примерах, на которые я не собираюсь ссылаться нигде больше, функции буду всегда обозначать одной буквой ***f***, а переменные - знаком ***x***.

Как увидим далее, ***defn*** - это не единственный способ объявления функций. Выражение в квадратных скобках ***[x]*** объявляет вектор (массив), элементы которого обозначены общим знаком ***x***, позволяющим далее ссылаться на элементы вектора. Очевидно, что данный вектор представляет аргумент функции ***f***. В Clojure аргументы функций всегда должны быть представлены в такой форме. Дальше располагается тело функции. Выражение

(/(apply + x)(count x))

указывает, что к элементам вектора применяются некоторые действия, определяемые рядом расположенным блоком, заключённым в круглые скобки. Текст в блоке расшифровывается так: результат, полученный

при вычислении выражения внутри первой пары скобок, делится на результат выражения во второй паре скобок. При этом выражение ***(apply + x)*** содержит встроенную функцию ***apply***, которая выполняет заданные действия над всеми элементами вектора. Знак + здесь не арифметическая операция, а тоже функция, которая является аргументом для ***apply*** и указывает, что элементы должны быть просуммированы.

Соответственно, ***(count x)*** содержит встроенную функцию ***count***, определяющую количество членов в переданном ей векторе (его размер). Следовательно получается, что функция ***f*** вычисляет среднее значение для всех элементов вектора.

Попутно мы установили интересный факт, заключающийся в том, что арифметическая операция деления в Clojure имеет префиксную форму, а именно: операция деления величины ***a*** на величину ***b*** записывается в такой форме:

(/ a b)

То-есть, операция деления имеет ту же форму, что и вообще любая функция двух переменных, например ***(fi x y)***. В Cloure все операции имеют такую форму, например: ***(+ a b)***, ***(* a b)*** и так далее.

Приведённый в примере код можно выполнить в Repl, в результате на экране появится такой текст:

#'user/f

который красноречиво говорит, что функция ***f*** готова к работе и значит мы её можем вызвать, передав ей какой-нибудь конкретный вектор, например:

(f [1 2 3 4 5]) → 3 - (Здесь и далее мы всегда будем пользоваться стрелкой → вместо слов получим, будет выведено, и тому подобное. Не будем также писать знак приглашения =>).

Я уже упоминал, что слово ***user*** является названием текущей области видимости (***namespace***). Это название присваивается по умолчанию при вызове Repl.

При вызове функции элементы вектора мы записали через пробел, но Clojure соглашается и на запятые; он их просто не отличает от пробелов:

(f [1, 2, 3, 4, 5]) → 3

В заданном векторе элементы представлены целыми числами и результат тоже оказался целым числом. Интересно, что мы получим, если среднее значение не равно целому числу, например:

(f [1 2 3 4]) → 5/2

Оказывается, Clojure умеет работать с рациональными дробями; компилятор даже автоматически упростил результат: вместо ***10/4*** выдал нам ***5/2***. Попробуем также ввести числа с плавающей запятой:

(f [1. 2. 3. 4.]) → 2.5

Программа работает нормально, кроме того, мы узнали, что не обязательно указывать нуль после десятичной точки. Кстати, элементы будут рассматриваться, как числа с плавающей запятой и в том случае, если среди них есть хотя бы только один такой элемент, а все остальные имеют форму целых чисел:

(f [1. 2 3 4]) → 2.5

А что будет, если ввести элементы типа текстовых величин, которые в Clojure, как и во всех других языках записываются в виде набора знаков в кавычках:

(f ["aa" "bb" "cc"])

Ничего не выйдет, так как операция деления для этого типа не имеет смысла; будет создано исключение (сообщение об ошибке), компилятор выдаст довольно длинный текст, который даже нет смысла здесь воспроизводить. Вообще, в подобных случаях, как правило, выводятся длинные и малопонятные тексты и это наследие от языка Java, для которого такие вещи очень характерны.

Можно также отметить, что мы нигде явно не указывали тип данных, а это значит, что Clojure способен определять (выводить, ***derived***) типы тех данных с которыми он работает и это важнейшее свойство дальше будет играть большую роль.

Программу легко изменить, например, для вычисления произведения элементов вектора:

```
(defn f [x]
      (apply * x))
(println (f [1. 2. 3. 4. 5.])) → 120.0
```

Здесь мы ещё ввели функцию вывода ***println***, при её вызове в Repl кроме результата, равного числу ***120.0***, будет ещё выведено слово ***nil.*** Дело в том, что в Clojure всё является выражением, возвращающим результат — последнюю вычисленную величину. В нашем примере на последнем месте стоит оператор вывода, который выводит текст, и ничего не возвращает. Поскольку это недопустимо, то во всех подобных случаях возвращается ***nil*** (ничто). Иногда этот ***nil*** приходится использовать для анализа ситуации.

Отметим ещё, что компилятору безразлично, как мы разбиваем текст на строки и делаем или нет отступы, которые нужны только для лучшей читабельности кода; всё определяется расположением круглых скобок. Нашу программу мы вполне могли бы написать и в одну строчку, если нравится (мне, например, это кажется хорошим вариантом):

(defn f [x] (/(apply + x)(count x))) (print (f [1. 2. 3. 4.])) → 2.5

Основным разделителем в тексте служит пробел, который часто (например в коллекциях) может заменяться запятой. Несколько пробелов подряд и перевод строки воспринимаются, как один пробел.

При вызове функции из библиотечного модуля применяется знак / вместо обычной точки. Имена модулей всегда с заглавной буквы:

(Math/sin 1.5) → 0.997

Обычно нельзя вводить дополнительные круглые скобки для улучшения читабельности кода:

(Math/sin (1.5)) → это вызовет исключение.

(Math/sin (+ 1.5 0.8)) → ***0.74570*** - но в такой ситуации скобки, конечно, необходимы

Упомянем ещё здесь о том , что Clojure позволяет напрямую вызывать методы языка Java, например:

(.toUpperCase "hello") → "HELLO"

Присутствие точки перед именем указывает на то, что метод ***toUpperCase***, переводящий знаки переданного методу текста в верхний регистр, является методом из библиотеки Java, а не функцией из библиотеки Clojure.

Итак, на небольшом примере мы узнали довольно много о синтаксисе и технике Clojure; дальше всё это мы рассмотрим подробно.

…..Базовые типы, коллекции

Clojure имеет обычный набор базовых типов: целые числа, числа с плавающей запятой, рациональные дроби, строки, булевы величины, ключевые слова, символы, регулярные выражения. Нет необходимости в подробном описании этих типов, их использование мы освоим на практических примерах. Отметим только, что строки комментариев в Clojure начинаются с точки с запятой (;).

Имеется ряд функций, преобразующих один тип данных в другой:
(float 77) → 77.0 - целое число переводится в число с плавающей запятой.
(int 45.8) → 45 - обратное преобразование выполняется без округления.
(str 68.023)→ "68.023" перевод числа в строковый тип, эта функция способна переводить в строковый тип величины любого типа.
(float "7.87") → ошибка, обратный перевод не работает.

Переменные типа ключевые слова записываются с двоеточием впереди (также, как в Ruby записываются так называемые символы): ***:a, :name*** и так далее.
Обычно ключевыми называют зарезервированные слова, но здесь это название применено потому, что такой тип чаще всего используется в качестве ключей в ассоциированных списках (хэшах). В качестве ключей хеша применяются ещё и идентификаторы с двойным двоеточием впереди - ***::a, ::name***. Тогда для элементов хэша будут генерироваться полные (квалифицированные) имена, но об этом позже.

Есть ещё тип символ (совсем не то, что в Ruby) - это любое выражение с одинарной кавычкой впереди:
'(+ 6 8) — такое выражение не воспринимается, как арифметическая операция, а рассматривается, как текст. В основном этот приём используется для «выключения» некоторых действий в программе. Если символ представлен идентификатором с кавычкой впереди, например, ***'name,*** то он указывает на имя объекта, а не на его содержание, как это было бы в отсутствие кавычки. Говорят, что символ вычисляется в себя самого:
user=> ***'alfa → alfa***
(***def s '(+ 6 8))*** - переменную ***s*** инициировали символом.
s → (+ 6 8) - текст возвращается без кавычки (В repl для получения значения переменной надо написать её имя без скобок). Стандартное слово ***def*** применяется дли объявления и инициации глобальных переменных; подробнее об этом позже.

Тип любого объекта в Clojure можно узнать с помощью команды:
(type name) - ***name –*** идентификатор объекта, например:
(type s) → clojure.lang.PersistentList
Снова получили громоздкое выражение, наследник Java Clojure выдаёт их постоянно; особо полезной информации они не содержат; в

данном случае существенно лишь слово ***List***, означающее, что ***s*** – список. Оказывается, что введённая выше переменная ***s*** представляет список ***(+ 6 8)***, о том, как вообще обрабатываются списки, речь будет идти далее.

(type 2.34) → java.lang.Double

В данном случае объект ***2.34*** имеет тип ***double***.

(def x 'aa)

(type x) → clojure.lang.Symbol

Значит, имеется и тип ***Symbol.***

Иногда выводится слово ***java***, а иногда ***clojure***. Это потому, что некоторые сущности принадлежат языку Clojure, а очень многие просто заимствуются из языка Java. Для программиста это, в общем-то, не имеет никакого значения. (Бесконечно повторяющееся бессмысленное слово lang из Java ужасно надоедает).

Кроме оператора ***type***, есть ещё оператор ***class,*** позволяющий получить ту же информацию:

(class s) → clojure.lang.PersistentList

…..Коллекции

Коллекции в Clojure играют основополагающую роль. Чаще всего предполагается, что коллекция содержит данные не какого-то конкретного типа, а что тип неопределённый (абстрактный). В действительности, в большинстве случаев элементы коллекции могут быть представлены любыми (смешанными) типами. Большинство встроенных функций могут работать с разными видами коллекций. Пока мы ограничимся рассмотрением только основных (базисных) коллекций.

Списки (***lists***)

Список записывается в виде перечисления, через пробел (но можно и через запятую), некоторых объектов (литералов) , заключённого в круглые скобки:

(yank hot foxt)

При вычислении первый элемент списка рассматривается, как функция, макрос (макроопределение) или как специальный оператор. Если ***yank*** - функция, то остальные элементы передаются ей в качестве аргументов. Если же ***yank*** – макрос или специальный

оператор, то использование остальных элементов определяется способностями самих макроса или оператора. В некоторых ситуациях используют списки с одинарной кавычкой впереди:
(def a '("shirt" "coat" "hat"))
a → ("shirt" "coat" "hat")
Эта кавычка указывает на то, что первый элемент в списке теперь не будет рассматриваться, как имя какой-то функции. Без этой кавычки инициировать переменную списком нельзя:
(def s (1 2 3)) → ClassCastException java.lang.Long cannot be cast to clojure.lang.IFn clojure.lang.Compiler$InvokeExpr.eval (Compiler.java:3648)
Это ещё один пример громоздкого сообщения. Все эти слова имеют конкретный смысл, но обычно он для программиста бесполезен. Существенно здесь только слово ***exception*** — исключение, которое вполне можно заменить словом ***ошибка***. Дальше я нигде не буду приводить эти нагромождения слов, которые на самом деле генерирует, на мой взгляд вполне идиотский язык, Java.

Вместо кавычки можно использовать встроенную функцию ***list***, превращающую набор своих аргументов в список:
(def a (list 1 2 3 4 5))
a → (1 2 3 4 5)

Списки могут содержать элементы разных типов, в том числе и другие коллекции, в частности и другие списки (вложенные). Приведём ещё примеры списков:
(5 2 7) - иногда только числа
() - пустой список, его можно заменить термином ***nil***.
(:fred 35) - ***:fred*** здесь ключевое слово, как они конкретно используются, узнаем позже.
(list 3 7 (list :a :b :c) 1 2) - содержит вложенный список.

Встроенная функция ***range*** позволяет генерировать списки с элементами, представленными числами из заданного диапазона. Например:
(range 5) → (0 1 2 3 4) - целые числа из диапазона ***0-5*** (конечная точка в список не входит).
(range 3 9) → (3 4 5 6 7 8) - диапазон может быть любым
(range 7 25 3) → (7 10 13 16 19 22) - можно задать шаг не равный ***1***
(range 3.7 19.4 3.5) → (3.7 7.2 10.7 14.2 17.7) - функция ***range*** способна генерировать числа типа ***float.***

Добавление, удаление и изменение элементов в списках реализуется в их начале (на левом краю). Функции ***cons*** и ***conj*** добавляют элементы в начало списка, но аргументы к этим функциям записываются в разной последовательности:
(def a (list 1 2 3))
(cons 5 a) → (5 1 2 3)
(conj a 5) → (5 1 2 3) - результат тот же.
Функции ***first*** и ***peek*** одинаковы и возвращают первый элемент (голову, head):
(first a) → 1
(peek a) → 1
Также одинаково работают и функции ***rest*** и ***next***, возвращающие исходный список без первого элемента (хвост, tail):
(r***est a) → (2 3)***
(next a) → (2 3)
Некоторая разница между этими функциями всё же есть, но она имеет значение только в достаточно специфичных обстоятельствах, с которыми мы ещё встретимся.
Функция ***pop*** аналогична этим функциям, только при применении к пустому списку она генерирует исключение, а ***rest*** и ***next*** – нет (они возвращают пустой список).
(pop a) → (2 3)

В Clojure списки в основном используются не для хранения данных, а для организации самого программного кода.

Векторы (vector)

Векторы — наиболее часто используемый вид коллекций. Они применяются в качестве списка аргументов для функций, в так называемых ***let***-формах, для хранения больших объёмов данных, для организации стека, при извлечении элементов из хэшей и так далее.

Векторы — последовательность элементов через пробел (или через запятую), заключённая в квадратные скобки — мы уже с ними познакомились. Все элементы вектора равноправны в том смысле, что вычисляются по порядку, один за другим. Если внутри вектора встречается список, то он обрабатывается как обычный список, то-есть по правилам списка. Вектор похож на массив и эти термины в данном случае равноправны. Пустой вектор имеет вид ***[]***. Извлечение элемента по индексу имеет такой синтаксис:

(def v [1 2 3 4 5]) - объявили вектор ***v*** и инициировали элементы
(v 2) → 3 - извлекли элемент с индексом, равным ***2***
Функция ***vec*** позволяет для создания векторов использовать другие коллекции:
(def f (vec (range 5)))
f → [0 1 2 3 4] - создали вектор из списка

В Clojure все коллекции — не изменяемые (***immutable***), но имеется функция ***into-array***, позволяющая создавать изменяемый (***mutable***) вариант векторов. Этот тип векторов полезен при обработке больших массивов данных.
(def ds (into-array [1 2 3]))
Такой вектор можно прочитать только командой ***seq,*** но нельзя вызвать как обычно по имени:
(seq ds) → (1 2 3)

Обратите внимание: большинство функций работают с векторами, а результат возвращают в виде списка. Если надо получить вектор, следует об этом позаботиться, например, применить функцию ***vec.***

Поскольку ***ds - mutable***, то его элементы можно изменять «на месте», например, с помощью функции ***aset***, изменяющей указанный элемент:
(aset ds 1 7) → 7 - возвращает вставляемое значение
(seq ds) → (1 7 3) - вектор ***ds*** изменился (не знаю, есть ли ещё функции, изменяющие такие массивы на месте).
Функция ***aset*** изменяет элемент по его индексу (в примере индекс равен 1). Однако, функция ***replace***, позволяющая изменять элементы по указанному их значению не изменяет исходный вектор:
(replace {3 9} ds) → (1 7 9) - изменили элемент, равный ***3***
(vec (seq ds)) → [1 7 3] - исходный вектор при этом не изменился, то-есть, ***replace*** не изменяет вектор на месте, а создаёт новый экземпляр вектора (здесь мы применили функцию ***vec*** и на выходе получили вектор).

Для добавления в существующий вектор новых элементов служит функция ***into:***
(def v [1 2 3])
(into ["a" "b"] v) → ["a" "b" 1 2 3] - элементы могут быть разных типов

Если первый аргумент функции ***into*** – вектор, то независимо от вида второго аргумента функция ***into*** возвращает вектор:

(into [:a :b] (range 5)) → [:a :b 0 1 2 3 4]
(into [1.2 0.99] #{1 2 3}) → [1.2 0.99 1 3 2] - здесь второй аргумент — множество (***set***)

Вектор может быть элементом любой коллекции. Например, комбинация функций ***map*** и ***vector*** (значит, слово ***vector*** ещё и представляет функцию) позволяет создать список из векторов:
(map vector [1 2 3] [4 5]) → ([1 4] [2 5])

Clojure позволяет создавать векторы с элементами, принадлежащими всем базовым типам: ***int, float, double, byte, short, boolean, char, symbol.*** Обычно компилятор определяет (выводит) тип из контекста, но есть и специальная функция ***vector-of***, позволяющая задавать тип элементов принудительно:
(into (vector-of :int) [Math/PI 2 1.3 0.25]) → [3 2 1 0]
Здесь все элементы вектора типа ***float*** приводятся к заданному типу ***int*** без округления (тип указывается всегда с двоеточием впереди). Функция ***vector-of*** используется в комбинации с функцией ***into***.
(into (vector-of :double) [1 2 3 '8]) → [1.0 2.0 3.0 8.0] - символ тоже приводится к ***double.***

Всего имеется три способа извлечения элементов из вектора:
(nth v 3) → 4
(get v 3) → 4
(v 3) → 4
Все они извлекают элемент по индексу, но все же между ними есть разница. При обращении к пустому вектору или при попытке извлечь элемент с не существующим индексом первый и третий способы генерируют исключение, а функция ***get*** возвращает значение ***nil***.
(get v 9) → nil

Функция ***seq*** в применении к вектору возвращает список:
(seq v) → (1 2 3 4 5)
И есть ещё функция ***rseq***, изменяющая порядок элементов в списке на обратный (осуществляет реверс):
(rseq v) → (5 4 3 2 1)

Функция ***assoc***, аналогично ***replace***, заменяет в векторе заданный элемент:
(assoc v 3 :dd) → [1 2 3 :dd 5]
При этом можно указать индекс на единицу больше конечного и тогда заданный элемент будет добавлен на конец вектора (на правый край):
(assoc v 5 "Hello") → [1 2 3 4 5 "Hello"]

Функция ***replace*** позволяет заменить сразу несколько элементов:
(def v [1 2 6 2 4 6 2 8])
(replace {2 :a, 6 true} v) → [1 :a true :a 4 true :a 8]

Функции ***get-in, assoc-in*** и ***update-in*** позволяют извлекать и заменять элементы вложенных векторов (двумерных матриц):
(def ar [[1 2 3] [4 5 6] [7 8 9]])
(get-in ar [1 2]) → 6 - извлекли элемент из второй строки третьего столбца
(assoc-in ar [1 2] 99) → [[1 2 3] [4 5 99] [7 8 9]] - заменили тот же элемент
Функция ***update-in*** позволяет одновременно с заменой элемента ещё и выполнить над ним заданное действие:
(update-in ar [1 2] * 5) → [[1 2 3] [4 5 30] [7 8 9]] - умножили этот же элемент на ***5***

Для реализации стека в Clojure лучше всего подходит вектор. Классический стек имеет два оператора: ***push*** и ***pop***. Для векторов аналогами этих операторов являются функции ***conj*** и ***pop***. Функция ***conj*** добавляет элемент, а ***pop*** – удаляет, в отличие от списков, на конце вектора (на правом краю). Поскольку векторы ***immutable***, то эти функции возвращают новый вектор, в отличие от классического стека с изменяемыми коллекциями.

Для считывания элемента из стека без его изменения применяется функция ***peek***:
(***def st [:a :b :c])***
(peek st) → :c
(pop st) → [:a :b]
(conj st :d) → [:a :b :c :d]
Можно сразу выполнить цепочку действий над стеком:
(def st [1 2 3])
(+ (peek st) (peek (pop st))) → 5

Обычно стек работает с большими объёмами данных, поэтому для него важна «затратность» операций в отношении времени. Все указанные функции имеют приблизительно одинаковую скорость работы. В Clojure есть и другие функции, выполняющие те же действия над векторами (***last, assoc, dissoc***), но они менее эффективны по затратам времени.

В принципе списки тоже можно использовать для организации стека. В частности, функции ***conj, pop, peek*** работают и со списками.

Разница лишь в том, что в этом случае голова стека будет находится не в конце, а в начале коллекции (на левом краю списка).
(def st (list 'a 'b 'c 'd))
(peek st) → a

Функция ***subvec*** позволяет получить срез вектора по заданным индексам начала и конца среза:
(def v (vec (range 10))) → [0 1 2 3 4 5 6 7 8 9]
(subvec v 3 7) → [3 4 5 6] - седьмой элемент не входит в состав среза
При извлечении среза из среза индексация промежуточная (не та, что у оригинала):
(subvec (subvec v 3 7) 1 3) → [4 5]

Map (hash)

Map – это то же самое, что в других языках называется «ассоциированный список», или ***hash***, или «отражение», или даже «словарь». Одна из основных функций для работы с элементами коллекций имеет идентификатор ***map***. Чтобы не путаться с терминологией и для краткости, дальше этот тип коллекций будем называть словом «хеш». Элементы хэшей представлены парами в виде ключей и значений. Ключ от значения отделяется пробелом, а между парами могут быть пробелы или запятые. Синтаксическая форма хеша – последовательность пар, заключённая в фигурные скобки, например:
{1 "one", 2 "two", 3 "three"}

Ключ играет ту же роль, что индекс в массиве; по ключу можно извлекать значения. Отличие только в том, что и ключи и значения могут иметь любой допустимый контекстом тип. Очевидно, что наличие нескольких одинаковых ключей не имеет смысла; на значения не накладывается никаких ограничений. В отличие от векторов, порядок вычисления пар не фиксируется и может быть разным в зависимости от обстоятельств. Пустой хэш - ***{}***.
Синтаксис извлечения элемента из хэша:
(***def m {"one""Marta","two" "Anna", "three" "Peter"})***
(m "three") → "Peter" - указывается ключ вместо индекса

Для входа в хэш часто используются векторы. Элементы хеша представлены парами ключ-значение и при извлечении такой пары, например с помощью функции ***first***, получим вектор с двумя элементами:

(first m) → ["one" "Marta"]
Можно проверить, является ли возвращаемая коллекция вектором. Для этой цели применяется функция - предикат ***vector?***
(vector? (first m))) → true - да, это так
Соответствующие функции есть и для коллекций других видов:
(map? m) → true - m – это ***хэш***

Посмотрим ещё на один характерный пример использования вектора для работы с хэшем:
Изменим хеш ***m***, использовав в качестве ключей ключевые слова:
(def m {:one "Marta", :two "Anna", :three "Peter"})
Теперь используем функцию ***doseq:***
(doseq [[d a] m] (println (str (name d) ":") a "!")) →
one: Marta !
two: Anna !
three: Peter !
Встроенная функция ***doseq*** организует цикл (применяется в основном с целью получения побочного эффекта). Она принимает на входе вектор ***[d a]*** и хэш ***m***. Элементы вектора ***d*** и ***a*** в цикле последовательно принимают значения пар - ключ и значение соответственно. Дальше в круглых скобках — тело цикла, в котором значения пар выводятся на печать. Здесь использована ещё функция ***name***, которая ключевое слово преобразует в строку, в результате чего слова ***one, two*** и ***three*** выводятся без предшествующего двоеточия. Функция ***str*** преобразует свои аргументы в одну строку.

В качестве ключей чаще всего используются ключевые слова, которые обеспечивают некоторые дополнительные возможности:
(def m {:a 1, :b 2, :c 3, :d 4})
В этом случае есть два способа извлечения значения по ключу:
(m :c) → 3
(:c m) → 3
То-есть, хеш является функцией своих ключей и, наоборот, ключи являются функциями своего хеша.

Множества *(set)*

Коллекция ***set*** в Clojure подобна множеству в математике: это совокупность не упорядоченных элементов без повторений (уникальных). Иногда ***set*** ещё называют наборами. Синтаксически множество — это последовательность разных объектов через пробел

(или запятую), заключённая в фигурные скобки, перед которыми пишется знак **#**:
(def s #{ 7 3 "two" :four 0x78})
s → #{7 "two" :four 3 120} **-** порядок элементов не выдерживается
Здесь ***:four*** – ключевое слово, а ***0x78*** – шестнадцатиричное число (при считывании автоматически преобразовалось в десятичную форму).
#{} **-** пустое множество.
Методы работы с ***set*** специфичны. Этот сорт коллекций используется не так часто.

Встроенная функция ***vec*** позволяет преобразовывать коллекции всех типов, в том числе и множества, в векторы:
(vec s) → [7 "two" :four 3 120]
(vec {1 "one", 2 "two", 3 "three"}) → [[1 "one"] [2 "two"] [3 "three"]]
(def a (list 1 2 3 4 5))
(vec a) → [1 2 3 4 5]

Множества являются функциями своих элементов, возвращающими сопоставимые элементы:
(def s #{:a :b :c :d})
(s :c) → :c
Если указанного элемента нет в множестве, возвращается значение ***nil.***
(s :e) → nil

Элементы множеств доступны через ***get*** – функцию:
(get s :b) → :b
(get s :t) → nil
Можно задать два аргумента и тогда поведение функции ***get*** может быть разным:
(get s :a :b) → :a - возвратился первый элемент
(get s :b :z) → :b - элемента ***:z*** нет в множестве
(get s :z :b) → :b - и в этом случае возвращается ***:b***
(get s :y :z) → :z - в множестве нет ни ***:y***, ни ***:z,*** получили на выходе ***:z***
Как видим, здесь мы фактически что задаём в качестве аргумента, то и получаем на выходе. В общем-то это не слишком интересно, но иногда, как увидим, бывает и полезным.

Для добавления элементов в ***set*** используется функция ***into***, а добавляемый элемент должен быть указан, как вектор:

(into s [4]) → #{4 :c :b :d :a}
Можно добвлять сразу несколько элементов:
(into s [4 5]) → #{4 :c :b 5 :d :a}
При попытке добавить существующие элементы, они игнорируются:
(into s [:b :c :f]) → #{:c :b :d :f :a}
Порядок расположения элементов всегда не предсказуем, но функция ***sorted-set*** позволяет упорядочить их расположение:
(sorted-set 6 5 2 9 5 1 3 5)→ #{1 2 3 5 6 9} - все дублирующие значения игнорируются

Добавить элементы в ***set*** можно и с помощью функции ***conj***:
(conj s 7 8 9) → #{7 :c :b 9 :d :a 8}

…..Функции и переменные

Функции в Clojure трактуются в более широком смысле, чем обычно: они могут играть роль переменных ***(vars)***, сохраняться в коллекциях, а также использоваться другими функциями в качестве аргументов и возвращаемых результатов. Синтаксически вызовы всех функций записывается в префиксной форме. Имеется, по крайней мере, два преимущества префиксной формы: можно передавать функции любое количество аргументов (при инфиксной форме только два) и обеспечивается единообразие вызовов функций, макросов, операторов. Если в тексте встречается открывающая круглая скобка, а затем следует какой-нибудь идентификатор, то он автоматически рассматривается, как имя функции, или макроса, или некоторой специальной формы (если перед скобкой не стоит одиночная кавычка, о чём уже упоминалось выше).

Мы уже встречались с арифметическими операциями. Приведём ещё несколько примеров:
(+ 1 2 3 4 5) → 15
Число аргументов может быть больше двух.
(/ 144 2 3 6) → 4
Заданная операция выполняется последовательно заданное число раз.
(* 1 2 3 4 5) → 120
Так можно вычислить факториал числа.

Переменные

Для объявления переменных используется служебное слово ***def:***
(def x 42) - ранее уже применяли.
Здесь переменная ***x*** инициируется значением ***42***; подобное объявление называют коренным связыванием ***(root binding)***. Можно считать, что это равнозначно операторам присваивания типа: ***x = 42*** или ***x := 42*** (или ***val x = 42*** в Scala) в других языках. После объявления в таком виде, переменная ***x*** будет доступна и не может быть изменена до конца программы; можно только объявить новую переменную с тем же именем. По сути ***x*** – глобальная переменная. Иногда, например в теле функции, может быть объявлена локальная переменная с тем же именем. Тогда локальная переменная временно (на время её жизни) маскирует глобальную. В некоторых ситуациях может потребоваться объявление переменной без инициации:
(def y).

Анонимные (не именованные) функции

Анонимные функции объявляются со служебным словом ***fn*** и не имеют имени. В остальном они не отличаются от обычных функций:
(fn [x y] (println "Hello") #{x y})
Эта функция выводит сообщение и возвращает результат в форме ***set*** с элементами, заданными в аргументах. Поскольку имени нет, эту функцию нельзя вызвать обычным способом, но есть возможность передать ей аргументы непосредственно:
((fn [x y] (println "Hello") #{x y}) "Anna" 25) → Hello #{25 "Anna"}
Значит, список переданных аргументов помещается сразу после блока с объявлением функции. Обратите внимание на то, что изменился порядок расположения элементов — порядок во множествах не фиксируется (произволен). Обычно анонимные функции называют функциональными выражениями. Указание аргументов после функционального выражения инициирует его выполнение.

В Clojure служебное слово ***def***, позволяет присваивать имена анонимным функциям. Можно также считать, что переменная инициируется анонимной функцией:
(def f (fn [x y] (println "Hello") #{x y}))
Функцию ***f*** можно вызывать, как обычно:
(f 25.078 true) → Hello #{true 25.078}

Присваивание имени таким способом по форме не отличается от объявления инициированной переменной, которую, по существу тоже

можно считать функцией без аргументов, возвращающей своё значение при вызове.

Другой (основной) способ объявления именованной функции с использованием служебного слова ***defn*** нам уже знаком. Фактически ***defn –*** это комбинация слов ***def*** и ***fn***. Функция имеет возможность принимать несколько самостоятельных блоков в своём теле. При вызове выполняется один из блоков, какой именно, определяется по переданному функции списку аргументов. Покажем на примере:

(***defn f ([x] #{x}) ([a b] (* a b)))***

(f "aaa") → #{"aaa"}

(f 2.5 3.0) → 7.5

Если при вызове мы передаём функции ***f*** один аргумент, выполняется первый блок (в примере этот блок принимает вектор и возвращает ***set***), если два аргумента — второй. Если передать функции больше аргументов, чем используется в блоках, генерируется исключение:

(f 3 4 8) → ошибка

Однако, существует возможность использовать список аргументов с переменным (неограниченным) их количеством. Для этого применяется знак ***&***:

(***defn f [x y & z] (vector x y z))***

Здесь ***& z*** можно рассматривать, как множество аргументов. Стандартная функция ***vector*** просто возвращает свои аргументы в форме вектора. Посмотрим, что получится при передаче функции ***f*** разного числа аргументов:

(f 1 2) → [1 2 nil]

Если ровно два аргумента, к вектору добавляется элемент, равный nil.

(f 1 2 3 4 5) → [1 2 (3 4 5)]

Если аргументов больше, чем явно указано, все «лишние» добавляются в виде одного элемента — списка. При передаче меньше, чем два аргумента, будет ошибка.

Есть ещё одна форма объявления функции с помощью анонимной функции с указанием списка передаваемых аргументов и с применением знака ***#***. Посмотрим на примере, как это выглядит и как используется:

(def f #(list %1 %2))

Стандартная функция ***list*** возвращает свои аргументы в форме списка. Здесь ***%1*** и ***%2*** не значения аргументов, а указатели на их

присутствие. В данном случае они указывают на то, что функция ***f*** может принимать два аргумента:
(f "a" "b") → ("a" "b")
В конце можно указать знак ***%&***, который будет означать переменное число аргументов.
(def f #(vector %1 %2 %&)) - теперь мы применили функцию ***vector***, возвращающую свои аргументы в виде вектора
(f 1 2 3 4 5) → [1 2 (3 4 5)] — как обычно, «лишние» члены группируются в список
Если в списке всего один аргумент, вместо ***%1*** можно писать просто ***%***:
(def f #(vector % %&))
(f 1 2 3 4 5) → [1 (2 3 4 5)]
Можно таким образом создавать разные функции:
(def f #(* %1 (+ %2 %3))) - это функция трёх аргументов
(f 5 1 3) → 20 - вызывается, как обычная функция
(def f #(* % (+ % %))) - а это уже функция одного переменного
(f 3) → 18

Блоки, глобальные и локальные переменные

Иногда необходимо несколько блоков объединить в один, более крупный блок. Для этой цели существует оператор ***do***, например:
(do (def x 5.2) (def y 1.3) (+ x y) (/ x y)) → 4.0
В блоке ***do*** все вложенные блоки выполняются последовательно друг за другом; возвращается в качестве результата, как обычно, последнее вычисленное значение. В нашем примере результат сложения ***(+ x y)*** хоть и был получен, но не использован и потерян безвозвратно. Объявленные в блоке с помощью ***def*** переменные ***x*** и ***y*** сохраняют своё значение (в функциональном программировании это называют побочным эффектом) и доступны для использования за пределами блока (в Repl для получения значения глобальной переменной надо написать её имя без скобок):
x → 5.2
y → 1.3
Ещё пример:
(defn f [x] (def y 5)(* x y))
(f 7) → 35

y → 5 - переменная ***y*** доступна во внешней области.

Ранее уже упоминалось, что Clojure позволяет использовать и локальные переменные не вызывающие побочного эффекта. Посмотрим на примере:

(let [r 5 pi 3.14 s (* r r)] (println "r is" r) (* pi s)) → r is 5 78.5

Для объявления локальных переменных используется слово ***let***, после которого в квадратных скобках указываются все эти переменные. Затем они могут использоваться в теле конструкции ***let*** (обычно называют ***let***-формой), но за его пределами невидимы (перестают существовать). Как видно из примера, при объявлении локальных переменных их можно инициировать (инициирующее значение просто пишется рядом через пробел, никакой знак присваивания не нужен). При инициации допустимы любые выражения, в том числе и использующие предыдущие переменные: выражение ***s (* r r)*** использует инициированную перед этим переменную ***r***. Для улучшения читабельности можно поставить запятые:

(let [r 5, pi 3.14, s (* r r)] (println "r is" r) (* pi s)) → r is 5 78.5

Тело конструкции ***let*** называют ещё неявным ***do***, поскольку здесь тоже вычисляются все блоки последовательно и возвращается результат последнего блока.

Сделаем ещё пару замечаний относительно техники применения анонимных функций:

(def f (let [x 2] (fn [y] (* y x))))

Здесь локальная переменная ***x***, созданная с помощью ***let*** -формы, инициирована целым числом ***2*** и используется в теле анонимной функции, а ***y*** является аргументом функции ***f***:

(f 3) → 6

Посмотрим теперь на такую функцию:

(defn f [n] (let [x n] (fn [y] (* y x))))

Аргумент ***n*** используется для инициации локальной переменной ***x***, а ***y*** – глобальная переменная, которая должна быть инициирована во внешней области. Что мы получим, вызвав функцию ***f*** с конкретным значением ***n***?

(f 5) – на выходе будет функция переменной ***y*** (в Repl будет выведена некая информация о внутреннем представлении этой функции). У нас есть две возможности инициации глобальной переменной ***y***. Во-первых, это можно сделать непосредственно:

((f 5) 7) → 35

а во-вторых, можно через ***defn*** объявить новую функцию двух переменных:
(defn g[n y] ((f n) y))
(g 5 7) → 35

Можно ещё присвоить имя функции одного переменного, полученной при заданном ***n***:
(def g (f 5))
(g 7) → 35
Но такой подход не слишком полезен.

Наконец, можно исключить из функции ***f*** локальную переменную ***x*** вместе с ***let***- формой, эту переменную создающей:
(defn f [n] (fn [y] (* y n)))
Эта функция ***f*** ничем не отличается от предыдущего варианта.

Как видим, Clojure имеет весьма гибкий механизм для реализации функций.

Предикаты (predicate)

Предикаты — это функции, возвращающие ***true*** или ***false***. В Clojure имена предикатов заканчиваются знаком вопроса ***(?)***.
(zero? 0.0) → true
(true? (> 3 8)) → false
В Clojure имеется много различных предикатов, для получения их полного списка имеется команда:
(find-doc #"\?$")
Приведём несколько примеров из этого списка:
clojure.string/includes? → ([s substr])
True if s includes substr.
Здесь указано, что предикат ***includes?*** (указывается полное имя) принимает два аргумента: строку ***s*** и подстроку ***substr***. Дальше идёт краткое описание результата; в данном случае предикат вернёт ***true***, если в строка включает указанную подстроку.
clojure.core/distinct? → ([x] [x y] [x y & more])
Returns true if no two of the arguments are =
Просто покажем пример:
(clojure.core/distinct? [1 2 3 4] [5 6 7]) → true
(clojure.core/distinct? [1 2 3 4] [1 2 3 4]) → false

…..Рекурсивные функции

В функциональном программировании единственным способом создания циклов без побочных эффектов является рекурсия — приём, когда функция в цикле вызывает сама себя. Для организации рекурсивных циклов в Clojure существует специальная функция ***recur***. Посмотрим на примере вычисления суммы ***1 + 2 + 3 + … + x*** для заданного ***x***:

(defn f [acc x](if (pos? x) (recur (+ acc x) (dec x)) acc))

Функция ***f*** принимает два аргумента: ***acc*** – задаёт начальное значение суммы (обычно называют аккумулятором) и ***x*** – заданное число. Функция ***recur*** организует повторный вызов для ***f*** с изменёнными аргументами: на место ***acc*** передаётся предыдущее значение ***acc***, увеличенное на значение переменной ***x***, а на место ***x*** – предыдущее значение ***x***, уменьшенное на единицу (функция ***dec***- так называемый декремент, уменьшает свой аргумент на единицу). Условная функция ***if*** организует окончание цикла — последующий за ней блок выполняется только до тех пор, пока блок ***(pos? x)*** возвращает ***true***. Встроенная функция ***pos?*** возвращает ***true*** только для ***x>0*** и, значит, при ***x = 0*** произойдёт выход из цикла. В качестве результата функция ***f*** возвращает значение аккумулятора ***acc***:

(f 0 5) → 15

Если в нашей программе заменить ***(+)*** на ***(*)***, получим факториал заданного числа, только при этом начальное значение аккумулятора надо принять равным единице:

(defn f [acc x](if (pos? x) (recur (* acc x) (dec x)) acc))
(f 1 5) → 120

Кроме ***if*** Clojure располагает ещё одним условным оператором — ***when***, который в некоторых случаях должен использоваться вместо ***if***. Проанализируем подробнее работу ***if*** на нашем примере. Оператор ***if*** имеет неявную ветвь ***else***, поэтому после блока, содержащего форму ***recur*** должен быть ещё один блок для этой ветви ***else.*** В нашем примере в случае, когда условие даёт ***true***, будет выполнен блок ***(recur (* acc x) (dec x))***, организующий повторный вызов функции ***f***. Если же получим ***false***, выполняется блок, идущий следом. В нашем случае этот блок имеет вид ***acc*** (для одиночной переменной скобки не нужны), который просто возвращает значение аккумулятора уже за

пределами цикла. Иногда за блоком, содержащим форму ***recur,*** никаких блоков нет и тогда оператор ***if*** зафиксирует ошибку. В такой ситуации вместо ***if*** надо применить ***when***, у которого нет ветви ***else***. Если условие даёт ***false***, оператор ***when*** выполняет выход из цикла с возвращением результата ***nil***. Если функция не возвращает результат, то она используется только ради побочного эффекта, каким является, например, вывод на печать. Приведём пример:

(***defn f [x] (when (pos? x) (print x " ")(recur (dec x))))***

(f 5) → 5 4 3 2 1 nil

Обратите внимание на то, что функции ***print*** передаётся два аргумента: переменная ***x*** и строковая константа — пробел. Оператор ***print*** самостоятельно приводит число к строковому типу и выполняет конкатенацию — замечательная вещь! Функция ***print***, в отличие от ***println***, не выполняет переход на новую строку.

Впрочем, Clojure позволяет создавать рекурсивные функции и без специального оператора ***recur***. Составим программу возведения числа ***x*** в степень ***n*** (есть библиотечная функция ***Math/pow***, наш пример только в учебных целях)

(defn pow [x n](if (zero? n) 1 (* x (pow x (dec n)))))

(pow 2 3) → 8

(pow 3.5 3) → 42.875

Вместо ***recur*** теперь просто используется повторный вызов функцией самой себя, в данном случае ***pow***, с передачей новых аргументов. Когда условие ***(zero? n)*** даст ***tru***e (переменная ***n*** достигнет нулевого значения), происходит выход из цикла. Подобная рекурсия строит такую цепочку вычислений:

(x * x * x *...* 1)

При большом ***n*** требуется много памяти для хранения всех сомножителей и при очень больших ***n*** может произойти переполнение стека. Это обычная проблема рекурсий. Она легко преодолевается применением аккумулятора, что мы и сделаем в последующих примерах.

Бывает, что требуется организовать рекурсивный цикл в теле некоторой головной функции. Для этой цели есть встроенная функция ***loop***. Снова напишем программу вычисления факториала, но такую, которая задаёт начальное значение аккумулятора, равное единице, автоматически:

(defn f [n] (loop [acc 1, x n] (if (pos? x) (recur (* x acc) (dec x)) acc)))

(f 5) → 120

Функция ***loop*** принимает вектор и при этом разрешает инициацию его элементов. Функция ***recur*** организует повторный вызов не головной функции ***f***, а вспомогательной ***loop***. Фактически роль функции ***f*** сводится к передаче заданного числа ***n*** функции цикла ***loop*** для инициации переменной ***x***.

Рассмотрим ещё один характерный пример организации цикла с помощью рекурсии. Функция ***rand-int*** генерирует случайные целые числа из заданного диапазона:

(let [x (rand-int 10)] x) → 8 -повторное выполнение этой формы будет выдавать случайные целые числа из диапазона ***0-10***.

Запрограммируем генерацию и вывод случайных чисел из заданного диапазона:

```
(defn f [x](print (rand-int 10)  " ")(if (pos? x)(recur (dec x))))
(f 5) → 7  0  4  7  3  5  nil
```

Функция ***rand*** генерирует случайные числа типа ***float***:

```
(defn f [x](print (rand 10.5)  " ")(if (pos? x)(recur (dec x))))
(f 5) → 9.2574  8.3250  5.4164  0.1306  10.4095  6.2366  nil
```

Кроме операторов ***def, fn, let*** и ***defn***, с которыми мы познакомились выше, есть ещё одна форма — ***letfn***, позволяющая создавать вложенные (локальные) функции. При этом головная (глобальная) функция имеет возможность вызывать вложенную функцию, передавая ей аргументы. Рассмотрим синтаксис формы ***letfn*** на примере генерации случайных чисел с выдачей их в виде списка или вектора:

```
(defn rnd [x d] (letfn
   [(f [x v](if (zero? x) v (recur (dec x)(cons (rand-int d) v))))]
     (f x [])))
```

Головная функция ***rnd***, принимает два аргумента ***x*** – количество случайных чисел и ***d*** – диапазон (начиная с нуля) их изменения. Функция ***rnd*** управляет следующим за ним в круглых скобках блоком ***letfn***. В блоке ***letfn*** объявлена локальная функция ***f***, которая вместе со своим телом помещается в квадратные скобки. За квадратными скобками следует вызов локальной функции ***f*** в обычной форме ***(f x [])***, где аргумент ***x*** передаётся из головной ***rnd***, а пустой вектор ***[]*** служит аккумулятором для случайных чисел накапливаемых в цикле. Вложенная функция ***f*** – обычная рекурсивная функция. В рекурсивном цикле новое значение ***x*** принимается равным старому,

уменьшенному на единицу функцией ***(dec x)***, а текущее значение вектора ***v*** заменяется на результат блока
(cons (rand-int d) v) в котором функция ***cons*** добавляет очередное случайное число в вектор ***v***. Выход из цикла происходит в момент, когда блок ***(zero? x)*** возвращает ***true***. При этом выполняется первая ветвь условия ***if – v***, возвращающая сформированный на этот момент вектор. Если ***x*** ещё не равен нулю, выполняется вторая (неявная) ветвь ***if***, реализующая рекурсию.
(rnd 10 100) → (99 15 68 18 1 64 87 3 65 89) – десять случайных чисел из диапазона ***0 -100***.

Как видим, возвращённый результат имеет вид списка (не очень ясно, почему так происходит), если нужен вектор, надо вместо возвращаемого ***v*** поставить ***(vec v)*** – функция ***vec*** коллекцию трансформирует в вектор.

Наконец, если требуются случайные числа типа ***float***, достаточно функцию ***rand-int*** заменить на ***rand***.

Недостаток программ на Clojure в большом количестве скобок, ухудшающих восприятие. Может быть следовало бы этот код записать в таком виде:

```
(defn rnd [x d]
  (letfn
    [
      (f [x v]
        (if
          (zero? x) v (recur (dec x)(cons (rand-int d) v))
        )
      )
    ] (f x [])
  )
)
```

Очевидно, что анализ расположения скобок в таком варианте выполнить легче, но читабельность кода вряд ли улучшилась.

Впрочем, всё это можно запрограммировать проще с помощью функции ***map***:

```
(defn rnd [x d] (map rand-int (repeat x d)))
(rnd 10 20) → (15 4 1 7 19 10 17 9 12 3)
```

Свойства функции ***map*** мы рассмотрим позже, а функция ***(repeat x d)*** создаёт вектор, имеющий ***x*** элементов, равных ***d***.

А вообще, в Clojure имеется много функций (директив) позволяющих создавать циклы; дальше мы с ними познакомимся.

…..Методы доступа к элементам коллекций

В документации Clojure операцию извлечения элементов из коллекций называют деструкцией (destructuring).

Рассмотрим эти методы на конкретных примерах. Объявим глобальную (***var***) переменную ***v*** и инициируем её заданным вектором:
(def v [42 "foo" 99.2 [5 12]])
Вектор ***v*** имеет 4 элемента разных типов. Есть несколько способов извлечения элементов:
(v 1) → ***"foo"*** - извлечение по индексу (нумерация начинается с 0). Этот способ показывает, что вектор — это функция своих индексов.
(first v) → ***42*** - функция ***first*** извлекает первый элемент (в функциональном программировании называется «голова» - ***head***)
(second v) → ***"foo"*** - соответственно, второй элемент
(last v) → ***[5 12]*** - последний элемент, у нас это вложенный вектор
(rest v) → ***("foo" 99.2 [5 12])*** - все элементы, кроме первого (в функциональном программировании называется «хвост» - ***tail***)
Функции ***first***, ***second, last*** и ***rest*** работают и с другими видами коллекций.
(nth v 2) → ***99.2*** - то же самое, что и ***(v 2)***
(***.get v 2)*** → ***99.2*** - то же самое, унаследован из Java

Покажем теперь несколько примеров использования этих методов в операциях над элементами вектора.
(+ (first v) (v 2)) → ***141.2*** - сумма первого и третьего элементов
(+ (first v) (first (last v))) → ***47*** - сумма первого элемента и первого элемента вложенного вектора

Коллекции часто применяются при инициации локальных (***local***) переменных, создаваемых с помощью ***let***-форм:
(let [[x y z] v] (+ x z)) → ***141.2***
Форма ***[[x y z] v]*** инициирует локальные переменные **x, y** и ***z*** тремя первыми элементами вектора ***v***. Остальные элементы при этом игнорируются. В результате приведённое выражение возвращает сумму первого и третьего элементов. Иначе это выражение можно ещё записать в таком развёрнутом виде:

(let [x (v 0) y (v 1) z (v 2)] (+ x z)) → 141.2
Скобки для извлекаемых значений здесь необходимы, иначе имеется очевидная неоднозначность. Приведём ещё один интересный пример:
(let [[x _ _ [y z]] v] (+ x y z)) → 59
Получили сумму первого элемента с элементами вложенного вектора. Поскольку второй и третий элементы не использованы, для них можно не вводить оригинальное имя, а использовать знаки подчёркивания. Обычно этот знак означает «какой угодно» или «любой» и называется групповым символом или ***placeholder*** (как это перевести?).

Можно также использовать уже знакомый знак ***&***, указывающий на множество элементов:
(let [[x & r] v] r) → ("foo" 99.2 [5 12])
Здесь форму ***[x & r]*** надо понимать, как первый элемент и все остальные, которыми инициируется переменная ***r***. Результат получен в форме списка.

Функция ***conj*** добавляет заданные элементы в конец вектора:
(def w (conj v "aaa"))
w → [42 "foo" 99.2 [5 12] "aaa"] - объявили и инициировали глобальный вектор ***w***.
Иногда требуется сохранить исходный вектор, изменяя вместо него копию. Для этой цели применяется директива ***:as***
(***let [[x _ z :as w] v] (def ww (conj w (+ x z))))***
ww → [42 "foo" 99.2 [5 12] 141.2]
w – локальная копия вектора ***v***, которую мы использовали для объявления глобального вектора ***ww***.

При использовании списков иногда приходится принудительно изменять порядок следования элементов (выполнять реверс). Для примера создадим функцию ***mp*** – аналог встроенной функции ***map***:
(defn mp [f c] (loop [co c, ac nil]
(if (empty? co)(reverse ac)(recur (next co)(cons (f (first co)) ac)))))
Функция ***mp*** получает два аргумента: функцию ***f*** (она выполняет заданные действия над элементами списка) и коллекцию — список ***c***. Анонимная функция в рекурсивном цикле тоже получает два аргумента: список ***co*** (инициирован исходным списком ***c***) и «аккумулятор» ***ac*** (инициирован пустым списком, вместо ***nil*** можно писать ***()***). При рекурсии первый аргумент заменяется на ***(next co)*** – это список ***co*** без первого элемента (здесь вместо ***next*** можно ***rest***), а

второй аргумент — на ***(cons (f (first co)) ac)*** — здесь функция ***f*** выполняет действия над первым элементом списка ***co***, а функция ***cons*** заталкивает этот результат в голову аккумулятора ***ac*** (на его левый край). Таким образом, в цикле в списке ***ac*** будет формироваться результат, в котором, однако, элементы представлены в обратном порядке. Условие ***(empty? co)*** даёт ***true***, когда список ***co*** будет пуст, при этом цикл прерывается, а все результаты будут в списке ***ac***. Для того, чтобы восстановить исходный порядок расположения элементов, перед возвращением результата список ***ac*** приходится реверсировать с помощью функции ***reverse.***

(mp - (range 5)) → (0 -1 -2 -3 -4)

В этом примере функция ***f*** представляет операцию ***(-)***, меняющую знак у элементов на обратный.

(mp #(* % %)(range 5)) → (0 1 4 9 16)

Теперь функция ***f*** возводит элементы в квадрат.

Если вместо списка применить в этом примере вектор, операция ***reverse*** не потребуется, так как функция ***conj*** заталкивает элементы в конец вектора (на правый край).

```
(defn mp2 [f c](loop [c c, ac []]
(if (empty? c) ac (recur (next c)(conj ac (f (first c)))))))
(mp2 #(* % %)(range 5)) → [0 1 4 9 16])
```

Обратите внимание, в этом варианте мы не стали вводить новое обозначение для локального вектора ***co***, а использовали тот же идентификатор ***c***. Теперь в нашем коде нет функции ***reverse***.

Рассмотренные методы работают и с другими видами коллекций, конечно, с учётом их специфики. Рассмотрим несколько примеров работы с хешами. Создадим коллекцию ***m*** с элементами разных типов:

(def m {:a 5 :b 6 :c [7 8 9] :d {:e 10 :f 11} "foo" 88 42 false})

В этом хэше ключи представлены ***:a, :b, :c, :d, “foo”, 42***, и соответствующие значения: ***5, 6, [7 8 9], {:e 10 :f 11}, 88, false***. Для нескольких ключей использованы «ключевые слова» вида ***:a, :b*** и так далее. Эта форма удобна для этой цели и применяется очень часто.

Теперь посмотрим на примеры извлечения значений из хеша ***m:***

(let [{x :a y :b} m] (+ x y)) → 11

Локальные ***x*** и ***y*** инициированы значениями, извлечёнными по ключам ***:a*** и ***:b***. Можно было и не вводить никаких локальных переменных, а получить результат непосредственно:

(+ (:a m) (:b m)) → 11

Поскольку ***(let [{x 42} m] x) → false***, то можно написать, например, такой код:

(let [{x 42} m] (if x (println "Anna") (println "Marta"))) → Marta

Напоминаю, что директива ***if*** имеет неявную ветвь ***else***, которая выполняется, если условное выражение возвращает ***false.*** Ещё один пример:

(let [{{x :e} :d} m] (* 2 x)) → 20

По ключу ***:d*** извлекается вложенный хеш ***{:e 10 :f 12}***, а затем по ключу ***:e*** извлекается из него значение ***10***.

Извлечём ещё элемент из вложенного вектора:

(let [{[x _ y] :c} m] (+ x y)) → 16

Из вектора ***[7 8 9]*** извлекаются первый и третий элементы, второй — игнорируется.

Подобным же образом можно использовать хеш и для извлечения элементов из вектора:

(def v [12 0 0 -18 44 6 0 0 5])

(let [{x 3 y 8} v] (+ x y)) → -13

Значит, выражение ***[{x 3 y 8} v]*** инициирует переменную ***x*** значением 4-го элемента вектора ***v***, а ***y*** – 9-го.

Функция ***assoc*** добавляет заданный элемент в хэш:

(def m {:a 2.3, :b 0.35, :c 12.77})

(assoc m :d 54.03) → {:a 2.3, :b 0.35, :c 12.77, :d 54.03}

Директива ***:as*** позволяет создавать копии для хешей, например, для того, чтобы при преобразованиях сохранить исходный хэш:

(***def m {:x 11 :y 22 :z 33})***

(let [{r1 :x r2 :y :as m1} m] (assoc m1 :sum (+ r1 r2))) → {:x 11, :y 22, :z 33, :sum 33}

Здесь ***m1*** является рабочей копией для ***m***.

Директива ***:or*** позволяет задать значение по умолчанию, если нужный элемент не будет найден:

(def m {:a 5 :b 7 :c 1})

(let [{k :u x :a :or {k 50}} m] (+ k x)) → 55 - Для инициации локальной переменной ***k*** предпринимается попытка извлечь элемент с ключом ***u***. Поскольку такой элемент отсутствует, принимается значение по умолчанию — ***50***.

(def m {:a 5 :u 7 :c 1})

let [{k :u x :a :or {k 50}} m] (+ k x)) → 12 - теперь такой элемент есть.

Есть ещё один вариант директивы ***or*** при котором он способен воспринимать значения ***true*** или ***false*** (в условных выражениях ***nil*** тождественен ***false***).
Объявим переменную ***k*** и инициируем её каким-либо значением:
(***def k 7)***
Теперь используем ***k*** в форме ***let*** с директивой ***or***:
(let [k (or k 50)] k) → 7 - значение ***k*** не равно ***nil***, значение по умолчанию отвергается.
(***def k nil)***
(let [k (or k 50)] k) → 50 - теперь принято значение по умолчанию.
Этим вариантом тоже можно воспользоваться для извлечения из хэша с заданием значения по умолчанию:
(let [{k :u x :a} m k(or k 50)] (+ k x))
Если ключа ***:u*** нет, то переменная ***k*** получит значение ***nil*** и произойдёт замена по умолчанию.

В том случае, когда ключи представлены ключевыми словами, при работе с хэшами эти слова приходится повторять, что при большом количестве элементов и при длинных словах сильно загромождает текст. Например:
(def chas {:name "Chas" :age 31 :location "Massachusetts"})
(let [{name :name age :age location :location} chas]
(format "%s is %s years old and lives in %s." name age location)) → "Chas is 31 years old and lives in Massachusetts."
Здесь ***let***-форма возвращает строку, сформированную с помощью функции ***format***. Эту строку затем можно передавать функции вывода. Знаки ***%s*** указывают, куда будут вставлены значения из расположенного далее списка переменных. В примере пришлось повторять слова ***name, age, location***. Имеется директива ***:keys,*** позволяющая исключить повторение:
(let [{:keys [name age location]} chas]
(format "%s is %s years old and lives in %s." name age location))
Результат будет тот же.

Рассмотрим теперь следующий пример:
```
(defn f [c] (cond
  (map? c) (seq c)
  (set? c) (map vector c c)
  :else (map vector (iterate inc 0) c)))
```

Функция ***f*** принимает коллекцию ***c*** произвольного типа. Работу встроенной функции ***cond*** можно грубо описать так:
(cond (условие) (блок))
Если условие даст ***true*** будет выполнен блок, в противном случае возвращается значение ***nil***. Функция ***map?*** возвращает ***true***, если ***c*** – хэш (***map***); а функция ***set?*** возвратит ***true***, если ***c*** – множество (***set***). Если не то и не другое, будет выполнена ветвь ***else***. Значит, в Cloure возможны и явные ветви ***else*** (пишется с двоеточием). Следовательно, функция ***cond*** способна принимать больше одного условия. Функция ***iterate*** реализует бесконечный цикл; в нашем случае в этом цикле происходит инкремент неявной переменной, начиная с нуля. При этом функция ***map*** тоже организует цикл по перебору элементов коллекции и в каждом проходе этого внешнего цикла функция ***iterate*** выполняет однократный инкремент. Вызывая функцию ***f*** с передачей ей коллекций разных видов, получим такие результаты:
(def v [:a 1 :b 2 :c 3 :d 4])
(def s #{:a 1 :b 2 :c 3 :d 4})
(def m {:a 1 :b 2 :c 3 :d 4})
(f v) → ([0 :a] [1 1] [2 :b] [3 2] [4 :c] [5 3] [6 :d] [7 4])
(f s) → ([1 1] [4 4] [:c :c] [3 3] [2 2] [:b :b] [:d :d] [:a :a])
(f m) → ([:a 1] [:b 2] [:c 3] [:d 4])

На основе нашей функции ***f*** создадим функцию ***pos***, позволяющую извлекать элементы из коллекций разных видов:
(defn pos [e c] (for [[i v] (f c) :when (= e v)] i))
Функция ***pos*** принимает целое число — индекс ***e*** и коллекцию ***c***. В этом примере использована не встречавшаяся ранее функция ***for***, которая напоминает классический оператор цикла ***for***, но имеет больше возможностей. Эта функция инициирует переменные ***i, v*** соответствующими значениями из пар, составляющих коллекцию — результат работы функции ***f***. При совпадении переменной ***v*** с заданным индексом ***e*** функция ***pos*** возвращает первый элемент пары.

Посмотрим на результаты использования функции ***pos***:
(pos 3 [:a 1 :b 2 :c 3 :d 4]) → (5) - напоминаю, что в этом случае коллекция, возвращаемая функцией ***f*** имеет вид:
([0 :a] [1 1] [2 :b] [3 2] [4 :c] [5 3] [6 :d] [7 4])
(pos 3 {:a 1, :b 2, :c 3, :d 4}) → (:c)
(pos 3 [:a 3 :b 3 :c 3 :d 4]) → (1 3 5)
(pos 3 {:a 3, :b 3, :c 3, :d 4}) → (:a :b :c)

Можно, конечно, поменять местами ***v*** и ***i***:
(defn pos [e c] (for [[i v] (f c) :when (= e i)] v))
(pos :d {:a 1, :b 2, :c 3, :d 4}) → (4)

Чтобы легче понять работу функции ***for***, приведу пример из документации:
(take 10 (for [x (range 10000) y (range 10000) :when (< y x)] [x y]))
([1 0] [2 0] [2 1] [3 0] [3 1] [3 2] [4 0] [4 1] [4 2] [4 3])
Здесь функция ***range*** не вычисляет всех своих элементов, ограничиваясь только нужным количеством в соответствии с принципом ленивых вычислений. О них мы поговорим позже.
Функция ***take*** повторяет вычисление своего блока заданное число раз.
В следующем примере функция ***for*** работает, как обычный цикл:
(for [x (range 5)] [x]) → ([0] [1] [2] [3] [4])

Модифицируем функцию ***pos***, расширив её возможности:
(defn pos [u c] (for [[i v] (f c) :when (u v)] i))
Теперь первый аргумент — условное выражение ***u***, которое позволяет отбирать элементы по заданному условию:
(pos #(not (int? %)) [:a 1 :b 2 :c 3 :d 4]) → (0 2 4 6) - если переменная ***v*** – не целое число.

…..Встроенные функции для работы с коллекциями

Clojure имеет огромное количество встроенных функций и разнообразных приёмов для работы с коллекциями всех видов. Некоторые мы уже видели, рассмотрим здесь другие.

Большинство структурных типов в Clojure можно рассматривать, как функции своих элементов. Уже упоминалось, что вектор — это функция своих индексов и можем вызвать вектор, как функцию, передав ей индекс как аргумент:
([3.7 "Hello" :k true] 3) → true - аргумент вектора индекс ***3***
В списках первый элемент рассматривается, как функция, а все остальные элементы — аргументы. Такая функция вернёт результат, если она синтаксически не бессмысленна. Форма арифметических операций на самом деле тоже список:
(* 5 7) → 35 - первый элемент — функция ****, 3*** и ***5*** — аргументы.

Поскольку допустимо элементы списка разделять также и запятыми, то это выражение можно записать и так:
(*, 5, 7) → 35 - компилятор согласен
Аргументов у арифметической функции может быть и больше двух:
(*, 2, 7, 3, 4) → 168
Но такая форма, конечно, не допустима для любых функций, у которых число аргументов чаще всего фиксировано.

Функция ***map*** позволяет работать сразу с несколькими или даже со всеми элементами вектора. Эта функция принимает два аргумента: некоторую заданную функцию и вектор, например:
(map dec [3 8 2 4]) → (2 7 1 3)
Функция — аргумент ***dec*** (декремент) уменьшает свой единственный аргумент на единицу. Функция ***map*** применила функцию ***dec*** ко всем элементам вектора последовательно и вернула результат в виде списка. В роли функции-аргумента может выступать любая функция одного переменного:
(defn f [x] #(* % x)) - эта функция умножает аргумент на заданное число
((f 4)7) → 28 - ранее мы с этой техникой уже знакомились.
(map (f 3) [3 8 2 4]) → (9 24 6 12) - все элементы умножаются на ***3***
Точно также работает ***map*** и со множеством (***set***):
(map (f 5) #{3 8 2 4}) → (20 15 10 40)
только порядок элементов в результирующем списке не соблюдается.
Функцию ***map*** можно использовать также для извлечения сразу группы элементов из вектора:
(map [3 8 2 4] #{1 3}) → (8 4)
Перечень нужных индексов задаётся в форме ***set***.

Подобным же образом функция ***map*** работает и с коллекциями типа хэш:
(map {:a 11, :b 34, :c 7} #{:a :c}) → (7 11)
В этом случае роль индексов играют ключи.

В Clojure не только коллекции могут выступать в роли функций, но и функции могут рассматриваться в качестве данных. Функции могут создаваться по мере необходимости, запоминаться в структурах (коллекциях), передаваться в качестве аргументов (это мы уже видели) и могут возвращаться другими функциями в качестве результата.

Встроенная функция ***comp*** позволяет создавать композиции функций, когда одна функция вызывает другую в качестве аргумента. Создадим три функции одного аргумента:

(defn f1 [x] (* x x))
(defn f2 [x] (Math/sin x))
(defn f3 [x] (Math/exp x))

Составим композицию из них (используется ***def***):

(def f (comp f1 f2 f3))
(f 2.5) → 0.14026

На самом деле здесь выполняется такая комбинация:

(f1 (f2 (f3 2.5))) → 0.14026

Встроенная функция ***cons*** добавляет в начало вектора элемент любого типа:

(cons 78.9 ["a" "b" "c"]) → (78.9 "a" "b" "c")
(cons [4.6 :g] ["a" "b" "c"]) → ([4.6 :g] "a" "b" "c")

Может добавить даже ***set*** и хэш:

(cons #{78.9 77.} ["a" "b" "c"]) → (#{77.0 78.9} "a" "b" "c")
(cons {:a 1, :b 2} ["a" "b" "c"]) → ({:a 1, :b 2} "a" "b" "c")

Результат всегда представлен списком. Функция ***cons*** может добавлять члены в ***set*** и в хэш, но результат иногда может оказаться неожиданным.

Функция ***repeat*** принимает два аргумента и возвращает список с повторяющимися элементами:

(repeat 3 "aa") → ("aa" "aa" "aa")

Функция ***vec*** принимает результат функции ***repeat*** и возвращает вектор:

(vec (repeat 3 "aa")) → ["aa" "aa" "aa"]

С помощью этих двух функций можно, например, создать функцию, возвращающую двумерный массив с одинаковыми элементами:

(defn f [h w] (vec (repeat h (vec (repeat w 0)))))
(f 3 4) → [[0 0 0 0] [0 0 0 0] [0 0 0 0]]

В Clojure имеется две функции ***max*** и ***min***, позволяющие выбирать одно значение из двух. С помощью этих функций и известной уже нам функции ***apply*** легко создать методы для отыскания наибольшего и наименьшего элемента в коллекции:

(def v [6.4 0.78 15.2 1.9])
(defn f [x] (apply max x))
(f v) → 15.2

Кажется, что здесь функции ***max*** передаётся только один аргумент, тогда, как ей полагается два. На самом деле функцией ***max*** управляет ***apply***. Замена ***max*** на ***min*** позволит находить минимальный элемент.

Практически также работает и ещё одна встроенная функция — ***reduce:***

(defn f [x] (reduce max x))
(f v) → 15.2

Как и в большинстве других языков программирования, в Clojure имеются встроенные средства сортировки коллекций. Функция ***sort*** способна сортировать векторы с элементами различных типов при условии, что все элементы взаимно соизмеримы.

(sort [1 5 7 0 -42 13]) → (-42 0 1 5 7 13)
(sort ["r" "z" "aa" "a"]) → ("a" "aa" "r" "z")
(sort [:z :aa :x :a]) → (:a :aa :x :z)
(sort [[3 2 1] [-1 2 1] [2 3 3]]) → ([-1 2 1] [2 3 3] [3 2 1])
(sort #{1 5 7 0 -42 13}) → (-42 0 1 5 7 13)

хэш сортируются по ключам.

Ну, и так далее. Как видим, функция ***sort*** сортирует по возрастанию, но это можно изменить, добавив условие сортировки:

(sort > [1 5 7 0 -42 13]) → (13 7 5 1 0 -42)

Вложенные векторы сортируются по первому элементу, но есть ещё одна функция сортировки — ***sort-by*** с дополнительным параметром, позволяющая управлять способом сортировки:

(***sort-by second [[:a 21 3] [:c 7 1] [:b 13 -4]]) →***
([:c 7 1] [:b 13 -4] [:a 21 3]) - по второму элементу и этим же приёмом можно отсортировать хэш по значениям

(sort-by last [[:a 21 3] [:c 7 1] [:b 13 -4]]) →
([:b 13 -4] [:c 7 1] [:a 21 3]) - по последнему элементу

Можно отсортировать и по убыванию:

(sort-by last > [[:a 21 3] [:c 7 1] [:b 13 -4]]) →
([:a 21 3] [:c 7 1] [:b 13 -4])

Если среди элементов есть несоизмеримые, получим исключение. Но есть возможность предварительно перевести все элементы в строковый тип с помощью функции ***str*** и после этого отсортировать:

(***sort-by str [85.2 "Bob" :dd]) → (85.2 :dd "Bob")***

Как видим, средства сортировки весьма разнообразны.

Обычно функции, выполняющие операции над коллекциями, возвращают результат в виде списка. С помощью функции ***vec*** (она отличается от ***vector***) можно «заставить» их возвращать вектор:
(vec (sort-by str [85.2 "Bob" :dd])) → [85.2 :dd "Bob"]

Функция ***select-keys*** позволяет извлечь из хэша один элемент:
(def m {:a 1, :b 2, :c 3})
(select-keys m #{:b}) → {:b 2}

Функция ***vals*** извлекает из хэша список значений в том же порядке:
(vals m) → (1 2 3)
Или так:
(vec (vals m)) → [1 2 3]
Функция ***map*** в сочетании с ***vals*** позволяет выполнить над значениями хэша нужные действия:
(def f #(Math/sin %))
(map f (vals m)) → (0.8414 0.9092 0.1411)

Функция ***zipmap*** позволяет выполнить над элементами нужные операции и вернуть результат в виде хэша с теми же ключами:
(zipmap (keys m) (map f (vals m))) → {:a 0.8414, :b 0.9092, :c 0.1411}

Более интересный пример, когда операции выполняются над элементами двух хэшей. Размер и ключи этих хэшей должны быть одинаковыми. Объявим ещё один хэш:
(def m1 {:a 5, :b 6, :c 7})
А также функцию двух аргументов:
(def f #(* %1 %2))
(zipmap (keys m1) (map f (vals m) (vals m1))) → {:a 5, :b 12, :c 21}

Функция ***merge*** соединяет два хэша в один:
(def m {:a 1, :b 2, :c 3})
(def m1 {:d 4, :e 5})
(merge m m1) → {:a 1, :b 2, :c 3, :d 4, :e 5}

Многие языки программирования позволяют создавать так называемые именованные аргументы функций. Такие аргументы можно передавать функции с указанием имён и без соблюдения порядка расположения в списке аргументов. Clojure тоже позволяет создавать такие аргументы. Рассмотрим на примере задачи о нахождении расстояния между двумя точками на плоскости. Обозначим точки ***p1*** и ***p2*** и зададим их координаты в форме хэша, в котором обозначения точек будут ключами, а координаты точек в виде

двучленных векторов будут значениями. То-есть, вид хэша будет следующий:

{:p1 [x1 y1], :p2 [x2 y2]}

Для объявления имён используется функция ***keys***. В целом, программа будет иметь такой вид:

```
(defn d [& {:keys [p1 p2]}]
(let [a (- (p2 1) (p1 1)) b (- (p2 0) (p1 0))]
(Math/sqrt (+ (* a a) (* b b)))))
```

Здесь ***&*** представляет неограниченное число аргументов, об этом уже говорилось ранее. ***let*** -форма использована для введения локальных переменных ***a*** и ***b***. При вызове функции ***d*** координаты точек вводятся с указанием имён ***p1*** и ***p2***, соответствующих ключам хэша, в произвольном порядке:

(d :p2 [6. 5.] :p1 [3. 1.]) → 5.0

В принципе функция ***d*** способна принимать любое количество точек:

(d :p2 [6. 5.] :p1 [3. 1.] :p3 [0 0]) →5.0

Результат тот же, третья точка в нашей программе не использована. Количество координат у точек тоже может быть любым:

(d :p2 [6. 5. 7.] :p1 [3. 1. 8.]) → 5.0

Третья координата в этом примере тоже не использована.

Имеется возможность вводить дополнительные условные ограничения, контролирующие данные, с которыми работает программа. Эти ограничения могут быть двух видов: ***pre-*** и ***postcondition*** (входные и выходные условия). Входные условия предназначены для анализа входных данных, а выходные — для результатов, возвращаемых функцией. Приведём пример функции с такими ограничениями:

```
(defn f [p1 p2]
      {:pre [(not= p1 p2)(vector? p1)(vector? p2)]
      :post [float? %]}
      (/ (- (p2 1) (p1 1))(- (p2 0) (p1 0))))
```

Из примера можно видеть синтаксис таких ограничителей. Условия записываются в фигурных скобках в виде хэша, где ключи — названия ограничителей, а значения - заключённые в квадратные скобки один или несколько условных выражений. Всё выражение в квадратных скобках должно возвращать ***true*** или ***false***, если условий несколько результат вычисляется с помощью логического ***and***. В нашем примере ограничитель ***pre*** получит ***true***, если ***p1*** не равно ***p2***

(логическая операция ***not=***) и оба аргумента представлены векторами. Ограничитель ***post*** получит ***true*** если функция вернёт результат типа ***float***. Вместо идентификатора для результата указывается знак ***%***. В случае не выполнения условий выводятся сообщения. Проверим:
(f [5 6] [5 6]) → (not= p1 p2) - векторы равны
(f [5 6] [1 2]) → (float? %) - результат — целое число
(f [5.0 6] [8 2]) → -1.3333 - все условия выполнены

Для работы с коллекциями есть очень полезная функция ***filter***, позволяющая отбирать элементы, удовлетворяющие заданному условию.
(filter even? (range 10)) → (0 2 4 6 8)
Здесь встроенная функция ***even?*** возвращает ***true*** для чётных чисел. Следовательно, функция ***filter*** принимает два аргумента: условное выражение и коллекцию.
(filter odd? [1 2 3 4 5 6 7 8]) → (1 3 5 7)
В предыдущем примере функция ***range*** генерирует список, а в этом примере использован вектор. Функция ***odd?*** возвращает ***true*** для нечётных чисел.

Условные выражения могут быть любыми, в частности, мы можем создавать их самостоятельно. Создадим такую функцию ***con***:
(defn con [x] #(> % x))
((con 3) 7) → true - эта функция сравнивает два числа по условию >.
Используем ***con*** для фильтрации списка:
(filter (con 7) [8 2 34 0 1 13 5]) → (8 34 13)
Ещё один пример:
(***defn di[d] #(zero? (rem % d)))***
Функция ***rem*** возвращает остаток от деления двух чисел:
(***rem 17 6) → 5*** - остаток от деления целых чисел
(rem 23.75 8.45) → 6.85000 - работает и с числами ***float***
Функция ***zero?*** возвращает ***true***, если аргумент равен нулю. Теперь используем нашу функцию ***di*** для фильтрации:
(filter (di 3) (range 20)) → (0 3 6 9 12 15 18) - отфильтровали элементы, делящиеся нацело на ***3***.
Можно, конечно, и не вводить специальную функцию — условное выражение, а скомпоновать всё вместе, создав функцию, принимающую коллекцию и число для проверки на делимость:
(defn fi [x v] (filter #(zero? (rem % x)) v))
(fi 3 (range 20)) → (0 3 6 9 12 15 18)

Такая форма, очевидно, более лаконична.

…..Макросы

Идея использования макросов реализована во многих языках программирования, но в Clojure этот полезный инструмент обладает более широкими возможностями. Макросы способны принимать некий программный код в виде текста в качестве аргументов, осуществлять над этим текстом какие-то преобразования и затем выполнять получившийся программный код. Фактически макросы предназначены для создания программистом нового синтаксиса Clojure. Рассмотрим, для начала, пример, не имеющий практического смысла, но иллюстрирующий работу макроса в простом варианте:

(require '(clojure [string :as st] [walk :as wl]))
(defmacro rev [cod] (wl/postwalk #(if (symbol? %) (symbol (st/reverse (name %))) %) cod))

В этом коде использованы две библиотечные функции: ***postwalk*** из библиотеки ***walk*** и ***reverse*** из библиотеки ***string*** (это не та функция ***reverse***, которая уже использовалась нами для реверса векторов). Оператор ***require*** открывает эти библиотеки, при этом следует обратить внимание на две особенности. Во-первых, перед списком библиотек должна стоять одинарная кавычка, а во-вторых, для библиотек должны быть созданы копии с помощью оператора ***:as***. Имена копий могут быть любыми, можно те же, что и у оригиналов.

Макрос объявляется полностью аналогично функции, только со служебным словом ***defmacro***. Далее следуют аргументы (у нас ***[cod]***) и тело макроса. Функция ***postwalk*** подобна уже хорошо знакомой нам функции ***map*** – выполняет над элементами коллекции заданные действия и возвращает полученную коллекцию в преобразованном виде. В нашем примере функции ***postwalk*** передаётся анонимная функция вида:

#(if (symbol? %) (symbol (st/reverse (name %))) %)

Условие ***if (symbol? %)*** выбирает для преобразования только элементы, представленные символами, а все прочие элементы (числа, скобки) пропускаются без преобразований. (Напоминаю, что символами в Clojure называют то, что в других языках чаще именуют литералами). Функция ***name*** преобразует символы в строки, функция

reverse меняет расположение знаков в строке на обратное, а функция ***symbol*** снова преобразует строку в символ. Таким образом, макрос ***rev*** в тексте ***cod*** выполняет реверс всех символов. Можно, конечно, анонимной функции дать имя и записать макрос в таком виде:
(def f #(if (symbol? %) (symbol (st/reverse (name %))) %))
(defmacro rev [cod] (wl/postwalk f cod))

Теперь передадим макросу такой бессмысленный код:
(qesod [gra (egnar 5)] (nltnirp (cni gra)))
Если в этом коде поменять порядок знаков в словах (в символах), то получим блок, выводящий на печать целые числа в диапазоне ***1...5***. Но, это как раз и умеет делать макрос ***rev***. Выполним его вызов:
(rev (qesod [gra (egnar 5)] (nltnirp (cni gra)))) → 1 2 3 4 5

Имеется функция ***macroexpand-1***, которая выводит тот код, который получается после преобразований, выполненных макросом:
(macroexpand-1 '(rev (qesod [gra (egnar 5)] (nltnirp (cni gra))))) → (doseq [arg (range 5)] (println (inc arg)))
(Тут надо не забывать ставить одинарную кавычку впереди). Действительно, из бессмыслицы получился разумный программный код.

Приведём ещё один вариант этой надуманной программы:
(defmacro rev [cod] (wl/postwalk #(if (string? %) (symbol (st/reverse %)) %) cod))
(rev ("qesod" ["x" ("egnar" 5)] ("nltnirp" ("cni" "x")))) → 1 2 3 4 5
(macroexpand-1 '(rev ("qesod" ["x" ("egnar" 5)] ("nltnirp" ("cni" "x"))))) → (doseq [x (range 5)] (println (inc x)))
Теперь все слова мы использовали в строковой форме.

Функция ***macroexpand-1*** является хорошим средством для отладки макросов, которая, вообще говоря, представляет значительные трудности. Для этой цели есть и ещё одна подобная функция ***macroexpand***, возвращающая более подробную информацию, хотя анализировать её тоже очень трудно.

Имеется «синтаксический сахар» (в Clojure много такого «сахара»), позволяющий в простых случаях с помощью макросов приблизить текст программы к обычной манере построения фраз, например, в английском языке. Для этого используется знак стрелки ***(->)***. Например:
(-> [1 2 3] reverse println) → (3 2 1)

Такой текст надо понимать так: взять вектор ***[1 2 3]***, выполнить в нём реверс и вывести результат на печать. В том, что это действительно макрос, а не что-то другое, можно убедиться, применив функцию ***macroexpand-1*** для получения того программного кода, который на самом деле выполняется:

(macroexpand-1 '(-> [1 2 3] reverse println)) →
(println (reverse [1 2 3]))

Как видим, макрос всё расставляет по своим местам. Тем не менее, при использовании этой техники можно получить много неожиданностей. Посмотрим не несколько более сложный вариант:

(-> (/ 144 12) (/ 2 3) str keyword list) → (:2)

Имея результат, можно понять, что здесь выполнялась такая последовательность арифметических операций: ***144/12/2/3 → 2.*** Функция ***str*** преобразует эту двойку к типу строка, ***keyword*** преобразует её к типу ключевое слово (добавляет впереди двоеточие), а ***list*** возвращает список. Проверим с помощью ***macroexpand-1:***

(macroexpand-1 '(-> (/ 144 12) (/ 2 3) str keyword list)) →
(list (keyword (str (/ (/ 144 12) 2 3))))

Действительно всё так, но согласитесь, что изначально результат был не очевиден. Попробуем ещё несколько больше усложнить задачу:

(-> (/ 144 12) (* 4 (/ 2 3)) str keyword (list :33)) → (:32 :33)

И вот какой код выполнялся на самом деле:

(list (keyword (str (* (/ 144 12) 4 (/ 2 3)))) :33)

Понятно, что если столько неожиданностей имеется в таких примитивных примерах, то в реально сложных задачах разбираться с этими макросами должно быть чрезвычайно трудно. И всё-таки макросы и интересны и могут быть полезны.

Отметим ещё, что макросы в форме стрелки удобны для тех случаев, когда несколько операций надо последовательно применить к одному и тому же объекту. Тогда этот объект ставим на первое место и затем перечисляем все эти операции в нужной последовательности — что может быть проще?

Есть ещё одна разновидность макросов этого вида, в которых применяется двойная стрелка ***(->>)***. Макросы данного вида применяются для обработки коллекций, последовательно применяя к ним перечисленные после стрелки блоки. Отличие от макросов вида ***(->)*** в том, что не создаются композиции функций. Посмотрим на примере:

(->> (range 10) (map inc) (reduce +)) → 55
В списке ***(range 10)*** функция ***map*** все члены увеличивает на ***1***, а затем функция ***(reduce +)*** вычисляет сумму всех членов. Раскроем код макроса:
(macroexpand-1 '(->> (range 10) (map inc) (reduce +))) →
(reduce + (map inc (range 10)))
Здесь всё очевидно. Применим теперь макрос ***(->)*** с этим же кодом:
(-> (range 10) (map inc) (reduce +)) - получаем ошибку
Посмотрим на раскрытый код:
(macroexpand-1 '(-> (range 10) (map inc) (reduce +))) →
(reduce (map (range 10) inc) +)
Как видим, полученная композиция функций не имеет смысла.

Функция quote и другие

Теперь немного отвлечёмся и рассмотрим ещё несколько очень своеобразных функций, имеющих отношение к макросам, а также применяемым и в любом другом коде. Речь прежде всего пойдёт о той одинарной кавычке, что уже встречалась нам несколько раз. У этой кавычки есть синоним ***quote*** и на самом деле это функция. Для анализа поведения ***quote*** воспользуемся встроенной функцией ***eval***, которая принимает в качестве аргумента выражение, вычисляет его и возвращает результат. Если аргумент представлен значением, то ***eval*** просто возвращает его:
(eval 2.75) → 2.75
Объявим какую-нибудь переменную и посмотрим, что делает с ней ***eval***:
(def x 88)
(eval x) → 88 - тоже возвращает значение.
Теперь попробуем применить ***eval*** к списку:
(eval (list 1 2 3)) → ошибка
Дело в том, что в списке, как мы уже знаем, первый элемент рассматривается в качестве функции с аргументами, представленными всеми остальными элементами списка. Функция ***eval*** пытается вычислить эту функцию, а единица из нашего примера не может быть функцией. А вот такой список работает:
(eval (list + 1 2)) → 3

Теперь применим к списку функцию ***quote*** (или '):
(eval (quote(list 1 2 3))) → (1 2 3)

Или, что одно и то же:
(eval '(list 1 2 3)) → (1 2 3)
В этом случае ***quote*** “выключает” вычисление и список рассматривается просто, как значение, которое и возвращается. По этой причине функцию ***quote*** называют ещё «подавляющей вычисление» (suppressing evaluation).

Попробуем объявить такую переменную:
(def a '(1 2 3))
и прочитать её:
a → (1 2 3) - похожа на список. Проверим:
(list? a) → true - действительно, список.
Значит, ***(list 1 2 3)*** и ***'(1 2 3)*** это одно и то же.

Можно, например, объявить такую переменную:
(def x '(+ 1 2))
И убедиться, что ***x*** – список и можно применить к нему функцию ***eval***:
(eval x) → 3

Применение ***quote*** к одиночному литералу возвращает символ:
(def x 'abc)
(symbol? x) → true

Кроме ***quote*** имеется ещё подобная функция, называемая s***yntax-quote***, которая синтаксически записывается, как обратная кавычка (`).
Опять попытаемся объявить переменную:
(def x `(1 2 3))
Посмотрим, является ли ***x*** списком:
(list? x) → false - нет, не список.
(***seq? x) → true*** - это ***seq*** – коллекция.
Применение ***syntax-quote*** к функции возвращает полное (квалифицированное) имя функции:
`range → clojure.core/range
Значит, функция ***range*** принадлежит библиотеке ***clojure.core*** — базовой библиотеке Clojure.
`(+ 1 2) → (clojure.core/+ 1 2)
То-есть, если применить ***syntax-quote*** к выражению, то будет возвращено это выражение, в котором для всех функций указаны полные имена.
(eval `(+ 1 2)) → 3
То-есть, получившееся выражение можно вычислять, как обычно.
Ещё пример:

`(map even? [1 2 3]) → (clojure.core/map clojure.core/even? [1 2 3])

Если ***syntax-quote*** применить к литералу, она вернёт символ с указанием соответствующей для него области видимости (по умолчанию user):

`f → user/f

Следующая функция из этой серии — это ***unquote***; синтаксически обозначается знаком (~). Функция ***unquote*** применяется под знаком ***syntax-quote*** и предназначена для приведения в действие вычисления выражения. На словах — сложно, на примере всё просто:

`(+ 10 (* 3 2)) → (clojure.core/+ 10 (clojure.core/* 3 2))

Здесь только указаны полные имена, но ничего не вычисляется. Но мы можем заставить что-нибудь вычислить в этом выражении:

`(+ 10 ~(* 3 2)) → (clojure.core/+ 10 6)

Можно таким способом вычислить любой блок:

`(+ 10 ~(* 3 (Math/sin 1.5))) → (clojure.core/+ 10 2.9924)

Однако, например такое выражение:

`(1 ~(2 3))

выдаст ошибку — тут нечего вычислять.

С помощью ***unquote*** можно, например, вставлять значения переменных в ***let***-формах:

(let [x 2] `(1 ~x 3)) → (1 2 3)

В этой ***let***-форме переменная ***x*** инициирована двойкой, а затем локальная переменная ***x*** в теле формы заменяется на своё значение с тем, чтобы вернуть список, как результат.

Ещё один пример:

(let [x '(2 3)] `(1 ~x)) → (1 (2 3))

Здесь переменная ***x*** инициирована с помощью ***quote*** списком ***(2 3)*** и этот список затем вставлен на место второго элемента результирующего списка.

Функция ***unquote*** имеет вариант под названием ***unquote-splicing*** с обозначением в виде сочетания двух знаков ***(~@)*** . Покажем, в чём она отличается от ***unquote*** на следующем примере:

(let [x '(2 3)] `(1 ~@x)) → (1 2 3)

Как видим, в отличие от предыдущего примера, в этом варианте вложенный список ***(2 3)*** «распаковывается» и его элементы просто добавляются в возвращаемый список.

Бывает, что нужно ввести в каком-то блоке или ***let***-форме уникальный идентификатор, нигде больше не используемый. Для

того, чтобы избавиться от необходимости придумывать такое имя можно использовать ***unquote*** в сочетании со знаком **#**, добавляемым на конце:

`abc# → abc__8275__auto__

В этом случае к введённому нами имени добавляется текст, автоматически генерируемый компилятором. При этом гарантируется, что подобное сочетание знаков нигде, кроме этого блока, не повторится.

Синтаксис макросов

Часто макросы возвращают код функции в виде списка. Списки удобны для использования в других функциях, специальных формах или в других макросах. Инструментом для формирования такого кода может служить функция ***list***. В простейшем случае это будет выглядеть примерно так:

(defmacro hello [name] (list 'prn name))

Здесь ***prn*** – это краткий синоним для функции ***println***. Кстати, функция ***print*** также имеет синоним - ***pr***. Поскольку идентификатор функции надо передать в результирующий список в неизменном виде, необходима ***quote***, а без неё ***prn*** будет рассматриваться, как литерал и не сможет быть распознан. Посмотрим, как будет выглядеть результат:

(macroexpand-1 '(hello "Marta")) → (prn "Marta")

Получили список, в котором первый элемент представляет функцию вывода.

Некоторые встроенные функции запрограммированы в форме макросов. Например, макросом является функция ***cond***. Сначала напомним, как она работает:

(***defn f [x] (cond (> x 5) (pr "> 5") (< x 5) (pr "< 5") (= x 5) (pr "= 5")))***

(f 7) → "> 5"

(f 5) → "= 5"

Функция ***cond*** принимает произвольное число пар условие — выражение. Если какое-то условие даёт ***true***, вычисляется соответствующее выражение и остальные условия не анализируются. Если ни одно из условий не будет выполнено, функция ***cond*** вернёт ***nil***.

Для анализа, с помощью команды ***source*** выведем текст макроса, реализующего функцию cond:

(source cond)

Для того, чтобы с полученным кодом можно было поэкспериментировать, я ввёл в него некоторые несущественные изменения, после которых получилось следующее:

```
(defmacro co [& c]
  (when c (list 'if (first c)
    (if (next c) (second c) (pr "exc"))
    (cons 'co (next (next c))))))
```

Изменения состоят в следующем:

- имя функции заменено на ***co***, иначе при попытке компиляции этого кода будет предупреждение о том, что функция с таким именем уже существует.
- удалён текст комментария, который выдаётся при получении справки с помощью команды ***doc***.
- для сокращения кода вместо запрограммированного исключения просто вставлен оператор вывода (***pr "exc"***).

В остальном код соответствует библиотечному.

Как видим, в данном случае оказалось недостаточно одной только функции ***list***; для формирования кода использована ещё и функция ***cons***. Попробуем разобраться, как данный код работает.

Макрос ***co*** принимает коллекцию ***c***, а знак ***&*** указывает, что в этой коллекции может быть произвольное количество членов. Функция ***when*** анализирует наличие аргументов. Если вызвать макрос ***co*** без аргументов, логически ***c*** будет восприниматься, как ***false***, и тогда ***when*** вернёт ***nil***. Если коллекция ***c*** не пустая, функция ***list*** формирует список, первый элемент которого ***if*** (мы уже отмечали, что в этом случае его надо записывать с ***quote***), а второй элемент — первый член коллекции ***c***, то-есть первое условие. Если это условие даёт ***true***, то выполняется ветвь, в которой с помощью ***if*** анализируется хвост коллекции ***c:*** (***next c*** – коллекция ***c*** без первого элемента). Если этот хвост отсутствует (***next c*** воспринимается, как ***false***), то выполняется (***pr “exe”***) и всё на этом заканчивается. Если же хвост не пустой, то вычисляется выражение для первой пары (***second c***), что и возвращается, как результат.

Если ***if (first c)*** даёт ***false***, то выполняется неявная ветвь ***else***, в которой задействована функция ***cons***. Напоминаю, что ***cons*** добавляет заданное значение в начало списка, например:

(cons 'co [1 2 3]) → (co 1 2 3)

Выражение ***(next (next c))*** – это коллекция ***c*** без первых двух элементов. Значит, вторая ветвь также формирует список, в котором первый элемент — имя макроса (поэтому здесь нужна ***quote***), а остальные элементы — коллекция ***c*** без первой пары. Таким образом, анализ начинается снова, теперь уже для второй пары. (Кстати, вместо двукратного применения ***next*** можно использовать функцию ***nnext***)

Посмотрим, какой код генерирует макрос ***co***:

(macroexpand-1 '(co (> x 5) (pr "> 5") (< x 5) (pr "< 5") (= x 5) (pr "= 5"))) → (if (> x 5) (pr "> 5") (co (< x 5) (pr "< 5") (= x 5) (pr "= 5")))

Здесь раскрыт только первый шаг — точно в соответствии с нашим анализом. Clojure имеет функцию ***macroexpand-all***, которая раскрывает все шаги подобного циклического анализа. Поскольку ***macroexpand-all*** принадлежит библиотеке ***walk***, то надо сначала эту библиотеку открыть:

```
(require '[clojure.walk :as w])
(w/macroexpand-all '(co (> x 5) (pr "> 5") (< x 5) (pr "< 5") (= x 5) (pr "= 5"))) → (if (> x 5) (pr "> 5") (if (< x 5) (pr "< 5") (if (= x 5) (pr "= 5") nil)))
```

Теперь имеем полностью раскрытый код макроса, представляющий собой последовательные условные выражения ***if***.

Применяются и другие средства для формирования кода. Рассмотрим ещё одну встроенную функцию - ***while***, которая запрограммирована тоже с помощью макроса. Посмотрим на её код:

```
(source while)
(defmacro whi [test & body]
      `(loop [] (when ~test ~@body (recur))))
```

Как и в предыдущем случае, здесь тоже изменено имя макроса и введены те же несущественные изменения.

Макрос ***whi*** принимает два параметра: условное выражение ***test*** и тело ***body*** в виде последовательности с произвольным числом членов. Смысл функции ***whi*** очень прост: до тех пор, пока условие ***test*** даёт ***true*** тело последовательно выполняется в цикле. Для организации цикла использованы встроенные функции ***loop*** и ***when***. В блоке под знаком ***syntax-quote*** с помощью ***unquote*** вставляются условие ***test*** и тело ***body***. Не вполне ясным может быть только использование ***splicing-unquote*** для раскрытия последовательности ***bod***y. Посмотрим на конкретный пример. Поскольку в этом примере использована

глобальная переменная ***x***, то её сначала объявим и инициируем нулевым значением:
(***def x 0)***
(whi (< x 5) (pr x) (def x (inc x))) → 01234
С помощью ***(def x (inc x))*** мы в цикле увеличиваем значение глобальной ***x*** на единицу. Цикл закончится, когда условие ***(< x 5)*** вернёт ***false***.
Теперь посмотрим на раскрытый код:
(macroexpand-1 '(whi (< x 5) (pr x) (def x (inc x)))) →
(clojure.core/loop [] (clojure.core/when (< x 5) (pr x) (def x (inc x)) (recur)))
Здесь тело цикла представлено двумя выражениями: ***(pr x)*** и ***(def x (inc x))***, которые выполняются последовательно.

Теперь попробуем убрать знак ***splicing-unquote***:
(defmacro whi [test & body]
`(loop [] (when ~test ~body (recur))))
И снова раскроем код:
(macroexpand-1 '(whi (< x 5) (pr x) (def x (inc x)))) →
(clojure.core/loop [] (clojure.core/when (< x 5) ((pr x) (def x (inc x))) (recur)))
Как видим, тело цикла оказалось в виде нераскрытого списка:
((pr x) (def x (inc x)))
и теперь первый элемент списка ***(pr x)*** рассматривается, как функция второго элемента ***(def x (inc x))***, что, конечно, не имеет смысла.

Хотя макросы во многих случаях полезны, при их использовании следует проявлять осторожность и обязательно проводить тщательный анализ. Иногда, казалось бы в простых ситуациях, могут иметь место неожиданности разного рода. Например:
(defn fun [x] (* x x))
(defmacro mac [x] (* x x))
Функция ***fun*** и макрос ***mac*** полностью идентичны и дают одинаковые результаты:
(fun 5) → 25
(mac 5) → 25
Попробуем использовать их с функцией map:
(map fun [1 2 3]) → (1 4 9) - всё нормально
(map mac [1 2 3]) - ошибка.

Итак, между функцией ***fun*** и макросом ***mac***, казалось бы неожиданно, обнаружилась разница. Дело в том, что макрос раскрывается на этапе компиляции, а на этапе выполнения программы (***runtime***) никаких макросов уже не должно быть. Однако, в нашем примере функция ***map*** вызывает ***mac*** как раз на стадии ***runtime***, что не допустимо. Вообще при передаче макросов другим функциям в качестве параметров практически всегда возникают проблемы:

(defn f [x] (fun x))

(f 5) → 25 - это нормально

(defn f [x] (mac x)) - это не проходит.

Рассмотрим ещё один пример типичной ошибки

(defmacro f [& body] `(let [x "hello"] ~@body))

Здесь всё для нас уже знакомо: в ***let***-форме локальная переменная ***x*** инициирована текстом ***"hello"***, а с помощью ***splicing-unquote*** раскрыт список тела ***body***. Попробуем конкретный вызов:

(f (pr x)) - ошибка

Мы всего лишь желаем вывести текст ***"hello"*** и получаем ошибку. Посмотрим на раскрытый код:

(macroexpand-1 '(f (pr x))) → (clojure.core/let [user/x "hello"] (pr x))

Оказывается, что локальная переменная ***x*** получила полное (квалифицированное) имя ***user/x***, а при передаче её в качестве параметра функции вывода ***pr*** – имя не полное. То-есть, ***syntax-quote*** не действует на параметр вложенной функции. Фактически мы имеем две разные переменные ***x***. Ситуацию в данном конкретном случае можно исправить так:

(defmacro f [& body] `(let [~'x "hello"] ~@body))

(f (pr x)) → "hello"

Не так легко понять, каким образом, но это работает.

(macroexpand-1 '(f (pr x))) → (clojure.core/let [x "hello"] (pr x))

Теперь переменная ***x*** в обоих случаях не получила полного имени.

Иногда подобную проблему можно решить за счёт уникальных идентификаторов, создаваемых совместным применением ***syntax-quote*** и знака ***#***. Пусть требуется к сумме элементов заданного вектора добавить случайное число:

(defmacro f [& n] `(let [x# (rand-int 10)] (+ x# ~@n)))

(f 1 2 3 4 5) → 18

Этот приём основан на том, что в одной и той же ***syntax-quote***-форме неоднократное использование ***x#*** будет давать одно и то же уникальное имя:

(macroexpand-1 '(f 1 2 3 4 5)) →
(clojure.core/let [x__8431__auto__ (clojure.core/rand-int 0)]
(clojure.core/+ x__8431__auto__ 1 2 3 4 5))

Можно найти бесчисленное количество всевозможных нюансов, связанных с макросами. По моему мнению, применять макросы следует только в том случае, когда это действительно даёт большую выгоду. И всякий раз следует проводить тщательный анализ и всестороннее тестирование.

…..Пространства имён (namespaces)

Пространство имён в Clojure приблизительно соответствует понятию пакета в других языках программирования. По умолчанию доступно namespace под названием ***user***. При входе в Repl появится приглашение в виде:

user=>

указывающее на то, что теперь активно это namespace. Для объявления нового namespace имеется две функции: ***in-ns*** и ***ns***. Например, мы можем объявить namespace под именем ***main1***, используя следующий синтаксис:

user=> (in-ns 'main1)

То-есть, имя namespace указывается с ***quote***. Можно, конечно, писать и так:

user=> (in-ns (quote main1))

Теперь приглашение будет иметь вид:

main1=>

Функция ***in-ns*** работает так. Если namespace с именем ***main1*** уже существовало ранее, то просто оно теперь становится активным. Если же такого namespace до этого не существовало, то оно создаётся заново и тоже становится активным. Значит, чтобы вернуться в ***user*** надо выполнить команду:

main1=> (in-ns 'user)
user=>

Объявим и инициируем каким-нибудь значением переменную в user:

user=> (def x "aaa")
Теперь перейдём в ***main1***:
user=> (in-ns 'main1)
и инициируем в этом namespace переменную с тем же идентификатором ***x*** новым значением:
main1=> (def x "bbb")
В ***main1*** переменная ***x*** доступна по имени:
main1=> x → "bbb"
Переменную из user теперь можно извлечь только по полному (квалифицированному) имени:
main1=> user/x → "aaa"
Синтаксис полного имени тот же, что и для библиотечных модулей, то-есть используется слеш.

Подобным же образом можно объявить и включить сколько угодно namespace с другими именами:
main1=> (in-ns 'main2)
main2=> (def x "ccc")
main2=> x → "ccc"
main2=> user/x → "aaa"
main2=> main1/x → "bbb"

Однако, нельзя объявить переменную или функцию в другом namespace:
main2=> (def user/x 123) – ошибка, так поступать нельзя.

Внутри любого namespace можно создать вложенное namespace:
user=> (in-ns 'main1.main2)
Находясь в ***user*** мы объявили ***main2*** внутри ***main1***. Ясно, что при этом ***main1*** уже должен был существовать. Обратите внимание на то, что имя вложенного namespace записывается через точку (не через слеш). Создадим и инициируем переменную во вложенном namespace:
main1.main2=> (def x :a)
Перейдём в ***use***r:
main1.main2=> (in-ns 'user)
и вызовем переменную:
user=> main1.main2/x → :a

Стандартные функции Clojure находится в ***core***, вложенном в ***clojure***, поэтому при обращении к ним из любого namespace, кроме ***user***, надо указывать полный путь:

main1=> (clojure.core/conj [5] 1 2 3) → [5 1 2 3]
Из ***user*** функции доступны по имени:
user=> (conj [:c] 1 2 3) → [:c 1 2 3]
Впрочем, можно и в этом случае указывать полный путь:
user=> (clojure.core/conj [77] 1 2 3) → [77 1 2 3]
Можно в core создавать новые функции:
main1=> (clojure.core/defn f [x]
(clojure.core/map #(clojure.core/* % %) x))
Но доступна эта функция по имени только из ***main1***:
main1=> (f [3 4 5]) → (9 16 25)
Из других namespace придётся указывать полный путь:
user=> (main1/f [3 4 5]) → (9 16 25)

Как уже указывалось, доступ к объектам в другой namespace возможен только по квалифицированному имени. Если таких обращений в другой namespace много, то можно их упростить, если воспользоваться встроенной функцией ***refer***.
main1=> (def x 555)
main1=> (in-ns 'main2)
main2=> main1/x → 555
Применим функцию ***refer*** к ***main1***:
(clojure.core/refer 'main1) → nil (она возвращает nil)
Теперь к объектам из main1 можно обращаться по имени:
main2=> x → 555
Однако, при этом надо иметь ввиду, что если теперь в ***main2*** попытаться объявить переменную с тем же именем ***x***, то будет конфликт имён.
main2=> (def x 777) - ошибка.

Есть и другой, более популярный, способ получения доступа к другим namespace с помощью директивы ***require***. Эту директиву мы ранее уже использовали неоднократно. Покажем ещё пример. В области ***clojure.set*** есть функция ***union***, позволяющая объединять множества. Использовать её можно так:
user=> (require 'clojure.set)
user=> (clojure.set/union #{1 2 3} #{4 5 6}) → #{1 4 6 3 2 5}
Можно избавиться от бесконечного написания слова ***clojure***, вводя синонимы:
u***ser=> (require '[clojure.set :as set])***
user=> (set/union #{1 2 3} #{4 5 6}) → #{1 4 6 3 2 5}

Функция ***all-ns*** позволяет узнать, какие namespaces открыты в данный момент.

Функция ***ns*** делает всё, что и функция ***in-ns.*** При этом имя области указывается без ***quote***:

user=> (ns main1)
main1=>

Кроме того функция ***ns*** имеет дополнительный набор директив: ***:exclude, :only, :as, :refer-clojure, :import, :use, :load, :require.*** Эти директивы позволяют выполнять более тонкую настройку для использования библиотек. Рекомендуется функцию ***in-ns*** применять при работе в Repl, а функцию ***ns*** – при создании компилируемых файлов. Впрочем, это совершенно не обязательно.

Работа с namespaces для устранения возможных конфликтов имён не представляет никаких трудностей, когда весь программный код расположен в одном файле. Другое дело, когда программа состоит из многих файлов, представляющих её составные части. Тогда приходится создавать проекты, используя leiningen или counterclockwise, формирующие комплекты из многих папок и файлов. В результате даже простейшая программа helloworld, состоящая из одной строчки, приведёт к получению пакета объёмом несколько мегабайт. Это обычный идиотизм Java. Разбираться со всеми хитросплетениями этих проектов здесь, я думаю нет смысла.

…..Записи (records)

Записи в Clojure использованы для реализации некоторых возможностей объектно-ориентированного стиля программирования (не очень ясно, почему принято такое название). Для объявления записи служит директива ***defrecord***. Познакомимся с записями на примерах:

(defrecord Person [name age pursuit]) → user.Person

Мы создали запись под названием ***Person***. Фактически при этом создаётся класс Java под этим именем, поэтому и название начинается с заглавной буквы, а если имя составное, то его рекомендуют записывать в CamelCase стиле (хотя это не обязательно). Аргументы ***name age pursuit*** можно считать полями класса. Запись является представителем нового типа данных, то-есть у нас теперь есть тип

данных ***Person***, который можно использовать на равных правах с базовыми типами. Класс предоставляет в наше распоряжение конструктор, позволяющий создавать экземпляры записи и передавать полям конкретные значения. Применяется следующий синтаксис:
(def p (Person. "Anna" 21 "teacher"))
Таким образом мы создали экземпляр записи с именем ***p*** (обращаю внимание на необходимость добавлять точку к имени записи). Есть и вторая форма вместо точки:
(def p (->Person "Anna" 21 "teacher"))
Проверим с помощью директивы ***class*** тип (или можно применять директиву ***type***):
(class Person) → ***java.lang.Class*** — действительно класс Java
(class p) → ***user.Person*** - переменная ***p*** имеет тип ***Person*** (выводится полное имя, то-есть с указанием пространства имён ***user***). Посмотрим на содержимое переменной ***p***:
p → #user.Person{:name "Anna", :age 21, :pursuit "teacher"}
Таким образом, запись имеет вид хеша, при этом имена полей преобразованы в ключевые слова. Доступ к значениям полей возможен по ключам, как обычно:
(:name p) → "Anna"
(:age p) → 21
Однако, вторая форма ***(p :name)*** для записей не допустима. Зато есть ещё и такая форма для извлечения значений:
(.name p) → "Anna"
Функция ***getBasis*** позволяет посмотреть список полей записи:
(Person/getBasis) → [name age pursuit]

Можно создавать сколько угодно экземпляров записи ***Person***:
(def p1 (Person. "Marta" 25 "engineer"))
(def p2 (Person. "Peter" 30 "doctor"))
и делать их членами коллекций:
(def v [p p1 p2])
С такими коллекциями можно работать, как с обычными, например применим к вектору ***v*** функцию ***map***:
(map :name v) → ("Anna" "Marta" "Peter")
Здесь из каждого элемента вектора ***v*** извлекается значение по ключу ***:name*** и результат возвращается в виде списка.

Поскольку запись на самом деле представлена хэшем, к ним применим весь набор функций: ***assoc, keys, get, seq, conj, into*** и другие.

(assoc p :adress "Moscow") → #user.Person{:name "Anna", :age 21, :pursuit "teacher", :adress "Moscow"}

Здесь мы в экземпляре ***p*** записи ***Person*** добавили новое поле ***adress***, инициированное значением ***"Moscow"***.

(keys p) → (:name :age :pursuit)

Однако, при этом поля экземпляра ***p*** не изменились (как и всё в Clojure, записи ***immutable***), при добавлении поля создаётся новый экземпляр:

(def p3 (assoc p :adress "Moscow"))

p3 → #user.Person{:name "Anna", :age 21, :pursuit "teacher", :adress "Moscow"}

Изменять поля записи и передавать им значения можно ещё и в такой нотации:

(map->Person {:name "Bob" :age 38 :pursuit "engineer" :adress "Vologda"}) → #user.Person{:name "Bob", :age 38, :pursuit "engineer", :adress "Vologda"}

Для того, чтобы посмотреть на примерах другие возможности работы с записями и не писать громоздких выражений, объявим запись с более короткими именами полей:

(defrecord Point [x y])

Для передачи полям значений можно применять функцию ***apply*** со стрелкой:

(apply ->Point [5 6]) → #user.Point{:x 5, :y 6}

Комбинируя функции ***map, partial, apply*** можно сразу создавать коллекцию из экземпляров записей:

(def col (map (partial apply ->Point) [[5 6] [7 8] [9 10]])) → #'user/col

col → (#user.Point{:x 5, :y 6} #user.Point{:x 7, :y 8} #user.Point{:x 9, :y 10})

Здесь ***col*** – список с элементами — экземплярами записей ***Point***.

(first col) → #user.Point{:x 5, :y 6} – извлекли первый элемент списка

(map map->Point [{:x 1 :y 2} {:x 5 :y 6 :z 44}]) → (#user.Point{:x 1, :y 2} #user.Point{:x 5, :y 6, :z 44}) - теперь передали значения и ввели новое поле для двух экземпляров ***Point***.

Имеются и другие приёмы работы с записями. Используя записи вместо, например, хешей можно получить ряд выгод, в частности,

скорость обработки записей выше, чем для хэшей. Вместе с тем в отличие от классических объектов ООП записи имеют только поля, но нельзя создать для них методов. В Clojure имеется механизм, позволяющий преодолеть этот недостаток. Для этого существуют так называемые протоколы (protocols). Главным назначением записей является их использование совместно с протоколами, что и позволяет реализовать некоторые принципы ООП. Отметим только, что вся эта техника пока довольно несовершенна, кроме того применяется она в тесном взаимодействии с инструментами Java, что, в общем-то обесценивает все те привлекательные возможности языка Clojure, с которыми мы познакомились. По этим причинам у меня нет желания рассматривать здесь работу с этими protocols.

В заключение этой темы отметим ещё, что новый пользовательский тип можно создавать с помощью директивы ***deftype:***
(deftype Point [x y]) → user.Point
Можно создать экземпляр типа Point:
(def p (Point. 2.5 0.73))
и извлекать значения полей:
(.x p) → 2.5
(.y p) → 0.73
Объекты этого типа имеют много общих свойств с записями, но есть, разумеется, и отличия.

…..Ввод — вывод

Операторы ***print*** и ***println*** служат для вывода на экран, с ними мы уже хорошо знакомы. Эти операторы имеют краткие синонимы ***pr*** и ***prn*** (применение кратких синонимов мне, например, всегда нравится).
(pr "Hello") → "Hello"nil
(prn "Hello") → "Hello"
nil
Встречались мы и с форматированным выводом, он не отличается от используемого в большинстве современных языков:
(def f "x = %8.3f")
(prn (format f 77.98745678)) → "x = 77,987" — почему-то вместо десятичной точки выводится запятая.

Ввод-вывод в файл выполняется с помощью модуля ***clojure.java.io,*** при этом требуется реализовать довольно сложную конструкцию, создающую связь между буфером и файлом. Проще, на мой взгляд, напрямую использовать средства Java.

Думаю, что для первого знакомства с языком этого достаточно. Некоторые аспекты, например такие, как обработка текста и регулярные выражения, я не стал затрагивать, поскольку в них нет ничего нового в сравнении с более ранними языками программирования, в частности, с тем же Java. В языке также реализован ряд более сложных идей, которых на этапе первоначального освоения языка лучше не касаться, а отложить «на потом».

В целом Clojure, конечно, оригинальный и интересный язык, пренебречь знакомством с ним непростительно. Есть смысл не ограничиться одним только знакомством, а попробовать использовать его в своих разработках. Самый большой недостаток языка, по моему мнению, это то, что реализован он на громоздкой, неповоротливой, неуклюжей (эпитеты можно продолжать) виртуальной машине Java. Если бы построить его на Си, такому языку «не было бы цены!»; уверен, что тогда он нашёл бы самое широкое применение.